School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Logic Programming Tools for Probabilistic Part-of-Speech Tagging

Joakim Nivre

October
2000

MSI
Växjö University
S-351 95 VÄXJÖ

Report 0062
ISSN 1400-1942
ISRN VXU/MSI/DA/E/--0062/--SE

Abstract

Part-of-speech tagging refers to the task of assigning lexical categories, or parts-of-speech, to words in a text, a basic task in many language technology applications. This task can be performed in many different ways, but one of the most widely used and best understood methods is to use a probabilistic model of the way in which parts-of-speech depend on word forms and on other parts-of-speech, typically represented as a Hidden Markov Model (HMM).

The purpose of this thesis is to develop a toolbox for probabilistic part-of-speech tagging, implemented as a set of logic programs that can be combined in different ways to build working systems, and designed in such a way that it can be used in teaching probabilistic part-of-speech tagging to students of computational linguistics or computer science. A secondary goal is to show that these tools can be optimized, both with respect to tagging accuracy and efficiency, in such a way that they may also be used to build taggers for practical applications outside the pedagogical context. In this way, we also hope to be able to show that logic programming is a viable alternative for the implementation of probabilistic language processing models, especially for prototyping purposes.

The report is organized as follows. Chapter 1 is introductory and defines the purpose of the thesis. Chapter 2 presents the necessary background material, divided into three sections. The first of these introduces the problem of part-of-speech tagging, some of its applications, and the most important methods used to solve the problem; the second discusses the probabilistic approach to part-of-speech tagging in more depth; and the third presents some central concepts of the logic programming paradigm. The basic toolbox, presented in chapter 3, covers tools for handling lexical and contextual models, processing input/output, and finding the optimal solution for a given input. Chapter 4 is devoted to optimization, with respect to accuracy as well as efficiency, and chapter 5 contains the main conclusions of the study.

Contents

Chapter 1

Introduction

Part-of-speech tagging refers to the problem of assigning lexical categories, or parts-of-speech, to words in a text, a problem that may appear rather uninteresting in itself but which is crucially important as the first step in many language technology applications, such as machine translation, information retrieval and grammar checking. There are many different approaches to this problem, but one of the most widely used and best understood methods is to use a probabilistic model of the way in which parts-of-speech depend on word forms and on other parts-of-speech, typically represented as a Hidden Markov Model (HMM).

This report describes the development of a toolbox for probabilistic part-of-speech tagging implemented in a logic programming framework. This toolbox is primarily intended for pedagogical use and is part of the resources for a web-based distance learning course in statistical natural language processing.[1] The pedagogical perspective provides the main motivation for two important design choices: the toolbox approach and the use of logic programming as the implementation vehicle for this approach. Let us therefore briefly discuss these two choices in turn.

1.1 Why a Toolbox?

Experience from teaching statistical natural language processing shows that it is very important to give students the opportunity to experiment with different ways of approaching the same problem in order to deepen their understanding both of the problem itself and of the concepts and tools involved in its solution. Even within a relatively narrow domain such as probabilistic part-of-speech tagging, there are many different choices to be made, with respect to probabilistic models, estimation methods, optimization techniques, etc. Moreover, these choices are to a large extent independent of each other, which means that they can be combined in many different ways. Whether we present the students with a fully implemented tagging system or ask them to implement such a system themselves, they will usually not get the opportunity to explore

[1] This web course, currently at `http://www.ling.gu.se/~nivre/kurser/wwwstat/`, was developed by the author with support from ELSNET (European Network in Language and Speech) and is now part of the ELSNET LE Training Showcase. It was also supported by the Socrates Thematic Network Project ACO*HUM (Advanced Computing in the Humanities) and is reviewed in [de Smedt *et al* 1999].

the whole space of different possibilities. One solution to this problem is the idea of a *toolbox*, i.e., a set of self-contained components that can be used to build a complete tagging system without too much time and effort, and where it is possible to combine the different components in all meaningful ways. This is the idea that we have tried to realize in the work presented in this report.

1.2 Why Logic Programming?

Probabilistic taggers are usually implemented in imperative programming languages such as C, mainly for reasons of efficiency. However, what we gain in speed we normally lose in transparency. And from a pedagogical perspective, transparency and clarity are of paramount importance. The declarativity of logic programs is very beneficial in making the probabilistic models more transparent and easier to understand. In many cases, definitions of key concepts can be expressed directly as logic program clauses, and alternative definitions of the same concept can be compared directly in the declarative logical framework.

Moreover, the logic programming paradigm is extremely well suited for implementing a flexible toolbox. Since the union of two logic programs is also a logic program, different components can be combined without the need to define special interfaces. As soon as we merge two logic programs by compiling them together, the general theorem prover will regard them as a single logic program, which means that predicates defined in one component may be freely used in the other component. In this way, logic itself provides a generic interface between all components, no matter how they are combined. In fact, in the compiled system there are no distinct components, or modules, in the strict sense, only a logic program compiled from multiple sources. However, in many cases, it is still useful to think of specific predicates as providing 'interfaces' between components, and of different components 'calling' each other, and we will often use this way of expression when it simplifies the presentation.

Finally, it is worth noting that, although the primary motivation for the present approach comes from pedagogical applications, we shall also see that, given suitable optimization techniques, probabilistic taggers implemented in a logic programming environment can be made reasonably efficient and may be used also in practical applications.[2] At any rate, the logic programming framework can be very useful in a prototyping stage, where it is essential to be able to experiment with different solutions in a quick and flexible manner. In fact, it is somewhat surprising that logic programming, which is the dominant paradigm in many areas of natural language programming, has so far had very little impact in the domain of probabilistic language processing. We suspect that part of the explanation lies in the fact that most probabilistic techniques in natural language processing have their origin in speech processing, which tends to favor more low-level implementation techniques.

[2] A subset of the tools presented in this report have been used to tag the 1.2 million word corpus of spoken Swedish collected at the Department of Linguistics, Göteborg University ([Nivre and Grönqvist forthcoming]).

1.3 Purpose of the Thesis

The primary purpose of this thesis is to develop a toolbox for probabilistic part-of-speech tagging, implemented as a set of logic programs that can be combined in different ways to build working taggers, and designed in such a way that it can be used in teaching probabilistic part-of-speech tagging to students of computational linguistics or computer science. A secondary goal is to show that these tools can be optimized, both with respect to tagging accuracy and efficiency, in such a way that they may also be used to build taggers for practical applications outside the pedagogical context. In this way, we also hope to be able to show that logic programming is a viable alternative for the implementation of probabilistic language processing models, especially for prototyping purposes.

1.4 Outline of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents the necessary background material, divided into three sections. Section 2.1 introduces the problem of part-of-speech tagging, some of its applications, and the most important methods used to solve the problem; section 2.2 discusses the probabilistic approach to part-of-speech tagging in more depth; and section 2.3 presents some central concepts of the logic programming paradigm. The basic toolbox, presented in chapter 3, covers tools for handling lexical and contextual models, processing input/output, and finding the optimal solution for a given input. Chapter 4 is devoted to optimization, with respect to accuracy as well as efficiency. Finally, chapter 5 contains the main conclusions of the study.

1.5 Availability of Source Code

The complete source code for the toolbox can be found in appendix C–H. It can also be downloaded from `http://www.masda.vxu.se/~nivre/logtag/`. All programs have been tested in SWI Prolog and SICStus Prolog but should run (possibly with minor modifications) in most Prolog environments.

Chapter 2

Background

This background chapter consists of three main parts. In the first part, we define the problem of part-of-speech tagging, discuss some of its applications, and briefly examine the different methods used to solve the problem. In the second part, we present the probabilistic approach to part-of-speech tagging in more detail. In the third and final part, we present some features of logic programming that will be important in the sequel.

2.1 Part-of-Speech Tagging

Let us begin by a simple example in order to illustrate what part-of-speech tagging is about. Consider the following somewhat artificial piece of text:

```
I can light a fire and you can open a can of beans.
Now the can is open, and we can eat in the light of
the fire.
```

The task of part-of-speech tagging consists in marking every word token in the text (including punctuation marks) with its correct word class, or part-of-speech. For the simple text above, this might produce the following result:

```
I/pn can/vb light/vb a/dt fire/nn and/cn you/pn can/vb
open/vb a/dt can/nn of/pp beans/nn ./dl Now/ab the/dt
can/nn is/vb open/jj ,/dl and/cn we/pn can/vb eat/vb
in/pp the/dt light/nn of/pp the/dt fire/nn ./dl
```

Here every word token is followed by a forward slash and a part-of-speech symbol, or *tag*, taken from a predefined inventory, or *tagset*. The number of tags in the tagset may range from about ten (corresponding to the parts-of-speech in traditional grammars) to several hundred, depending on the level of detail needed for a particular application. In what follows, we will rarely be concerned with the details of the tagset, but for examples we will adopt the basic tagset used in the Stockholm-Umeå Corpus (SUC), which is described in appendix A (see also [Ejerhed *et al* 1992]). It should also be noted that the slash notation used in the example above is just one of many different formats that can be used for tagged output, although we will not discuss output formats in this report.

2.1.1 Applications of Part-of-Speech Tagging

Before we go on to describe the problem of part-of-speech tagging in more detail, it may be worth saying something about its practical applications. Broadly speaking, we can divide these applications into three classes, depending on the depth and nature of the linguistic analysis of which tagging is a part:

1. First of all, there are situations where part-of-speech tagging in itself is the end-point of linguistic analysis. This is usually the case, for example, when part-of-speech tagging is used to improve precision in information retrieval by disambiguating search terms, or when part-of-speech information is used to improve language modeling in automatic speech recognition systems. This is also true in most cases for the annotated linguistic corpora that are used as resources for language research and for the development of different language technology applications.

2. Secondly, part-of-speech tagging may be the first step of a more full-fledged syntactic analysis, usually involving a grammar-based parser. This kind of analysis may be found, for example, in grammar checkers.

3. Thirdly, part-of-speech tagging may be preliminary not only to syntactic but also to semantic analysis, as in different kinds of language understanding systems, ranging from information extraction systems to dialogue systems, and in machine translation systems.

This list is by no means exhaustive but should at least give an impression of the wide range of applications where part-of-speech tagging is a relevant task.

2.1.2 Defining the Problem

Let us now try to define the problem of part-of-speech tagging a little more precisely. Given a sequence of word forms $w_1 \cdots w_k$, the problem consists in finding a corresponding sequence of word classes $c_1 \cdots c_k$, where c_i is the word class of w_i (for $1 \leq i \leq k$).[1] The *accuracy* of a particular tagging $c_1 \cdots c_k$ for a word sequence $w_1 \cdots w_k$ can be measured as the percentage of tags c_i that are in fact the correct tag for the corresponding word w_i. The *accuracy rate* of a tagging system can be defined as the average accuracy produced by that system when applied to a representative sample of input sequences.[2]

Why should we expect less than 100% accuracy in part-of-speech tagging? There are two main reasons, one theoretical and one more practical. The theoretical reason is that natural languages contain a large number of ambiguous word forms, i.e. word forms that may have more than one grammatical category. For example, if we return to our simple example text, we may note that *can* is sometimes a verb (`vb`) (as in *I can light a fire*) and sometimes a noun (`nn`) (as in *a can of beans*). Similarly, *light* can be a verb or a noun (or an adjective,

[1] The terms *word class*, *part-of-speech* and *tag* will be used interchangeably in the following. However, we will consistently use the symbol c [for *class*] to denote classes/parts-of-speech/tags.

[2] It should be noted that other definitions of the tagging problem are possible. For example, in larger systems it is sometimes convenient to allow the tagger to assign more than one tag to each word and leave the final decision to some other module in the system. We will ignore this complication in the sequel.

although this possibility is not exemplified in the text), and *open* can be a verb or an adjective. This means that even if we have a list of all the words in the language, with their possible parts-of-speech (a *lexicon*), this does not give us enough information to assign the correct tag to a given word token in the context of a sentence. Therefore, we must somehow exploit the information given in the context of occurrence to choose the correct tag for words that have more than one possible word class. This is usually referred to as the *disambiguation problem* in part-of-speech tagging.

The second, more practical reason why we can expect errors in part-of-speech tagging is that, in practice, we never have a complete list of all the words in the language with their possible parts-of-speech. No matter how large our lexicon is, if the tagger is supposed to work on unrestricted text, we must always expect to come across words that are not in the lexicon and for which we have no information about their possible parts-of-speech. Such words are normally said to be *unknown*, and the problem of correctly analyzing them is known as the *unknown word problem* in part-of-speech tagging.

The unknown word problem can in fact be regarded as a special, rather extreme case of the disambiguation problem, if we consider unknown words as words that can have any part-of-speech in the current tagset. Normally, however, we will try to reduce the unknown word problem by restricting the set of possible tags to *open classes* such as nouns and verbs. These classes, which typically contains thousands of members, are opposed to *closed classes*, such as conjunctions and prepositions, which rarely contain more than a hundred members and where we can therefore be more confident that the lexicon is complete.

Finally, it is important to note that unknown words affect the accuracy of tagging in two different ways. First, there is the problem of assigning tags to the unknown words themselves. This is again done mainly on the basis of contextual information. For example, if the preceding words are *the old*, we can be fairly confident that the unknown word is a noun.[3] The second way in which unknown words affect accuracy is that they reduce the contextual information needed to disambiguate other, known words, which means that they can cause errors also in the analysis of neighboring words.

2.1.3 Approaches to Part-of-Speech Tagging

Having defined the problem of part-of-speech tagging and discussed some of the difficulties involved, we are now in a position to give an overview of different approaches to the problem. This overview will necessarily be rather brief, but in the next section we will give a more in-depth presentation of the probabilistic approach with which we are primarily concerned in this report.

The wide variety of methods used to perform part-of-speech tagging can be classified in many different ways. It is customary to make a broad division into 'statistical methods' and 'rule-based methods'. However, this classification is a bit misleading since many of the so-called rule-based methods rely on statistics for knowledge acquisition (e. g., [Brill 1992]) and some of the 'non-statistical' methods are not really rule-based in the traditional sense (e. g.,

[3] That there is nevertheless a chance of error is shown by a classic example from the literature on syntactic parsing, *the old man the boat*, where *man* must be interpreted as a verb in order for the sentence to be interpretable syntactically.

[Daelemans *et al* 1996]).

Here we will instead begin by discussing what is common to all or most of the different approaches, namely the kind of linguistic information that they are based on. Then we will classify the different methods with respect to two different parameters:

- How is the linguistic information used in tagging?
- How is the linguistic information acquired?

Virtually all tagging methods rely on two different kinds of linguistic information, which we may call *lexical information* and *contextual information*, respectively. Lexical information (in the context of part-of-speech tagging) is information about what word forms exist in a given language, and what parts-of-speech are possible for a given word form. For example, *can* is an English word form, which (given the simple tagset in appendix A) can occur with the tags `vb` (verb) and `nn` (noun), although the former is much more common than the latter. Contextual information is information concerning the tendency for different parts-of-speech and/or word forms to occur together. For example, in English determiners (such as *a* and *the*) are very rarely followed by finite verbs (such as *runs* and *talked*).

Although both lexical and contextual information are exploited by any good tagging system, the way it is represented and put to use in actual tagging may vary a lot. Broadly speaking, we can distinguish three main approaches with respect to the actual tagging process:

- The bold approach: 'Use all the information you have and guess!' This strategy is primarily associated with probabilistic tagging ([DeRose 1988, Cutting *et al* 1992, Merialdo 1994, Nivre 2000]), but is also found in approaches based on memory-based learning ([Daelemans *et al* 1996]) and neural networks ([Schmid 1994]). What is common to these approaches is that all the information available to the tagger is combined in order to select the best solution in one go. What differs is the way the linguistic information is represented and the kind of evaluation metric used to select the best candidate.
- The cautious approach: 'Don't guess, just eliminate the impossible!' This strategy is represented by Constraint Grammar ([Karlsson *et al* 1995]), as well as Finite State Intersection Grammar ([Koskenniemi 1990]), and works by first tagging every word with all its possible tags and then systematically removing tags that are ruled out by contextual constraints (such as the constraint that determiners are not followed by finite verbs). Whatever remains when all the contextual constraints have been applied is considered the correct analysis. However, this means that, in principle, a word may end up with more than one tag or indeed no tag at all (although the latter possibility is usually ruled out by a general constraint to the effect that the last tag cannot be removed). This strategy therefore may produce good recall (because it avoids guessing with insufficient information) but at the cost of remaining ambiguity.
- The whimsical approach: 'Guess first, then change your mind if necessary!'. This strategy, which is primarily represented by the transformation-based approach of [Brill 1992], is a two-step approach. In the first step, the

tagger nondeterministically assigns one tag to each word. In the second step, a set of transformation rules that change tags according to contextual conditions are applied in sequence. Whatever remains after all transformation rules have applied is considered the correct analysis. In a way, this is a hybrid of the two preceding strategies. The first step, although in principle nondeterministic, is usually implemented in such a way that every word is assigned its most frequent part-of-speech (so that *can* would get the tag `vb`), which can be seen as the 'best guess' in the absence of contextual information. In the second step, transformation rules apply very much in the same way as constraints, except that tags are replaced rather than removed (so that the tag for *can* is changed from `vb` to `nn` if the preceding tag is `dt`).

With respect to the way in which linguistic information is acquired, the different tagging approaches can be divided into two groups:

- Machine learning: Most of the approaches discussed above use some kind of machine learning to automatically acquire the necessary information from linguistic corpora. This is true for probabilistic tagging, memory-based tagging, neural network tagging and transformation-based tagging. Learning can be either supervised (learning from pre-tagged corpora) or unsupervised (learning from raw text data), although supervised learning tends to produce better results.

- Hand-crafting: Manually creating the rules or constraints needed for tagging was more common in the early days of natural language processing. However, some of the most successful tagging systems available, such as those based on Constraint Grammar, still rely on hand-crafted constraints, although the possibility to induce such constraints automatically is also being investigated (see, e.g., [Lindberg and Eineborg 1998, Lager 1999]).

The accuracy rates for different tagging methods are fairly comparable and typically range from 95% to 98% depending on tagset and type of text, although the comparison of different methods is complicated by the fact that not all methods implement the condition of forced choice (one tag per word) that we have presupposed in our presentation so far. However, it also seems that different kinds of taggers make different kinds of errors, which suggests that performance can be further improved by combining different approaches (see, e.g., [van Halteren *et al* 1998]).

2.2 Probabilistic Part-of-Speech Tagging

After this brief introduction to the problem of part-of-speech tagging and some of the different methodological traditions, we will now focus exclusively on the probabilistic approach, which uses a probabilistic model during the actual tagging phase and try to find the most probable part-of-speech sequence for a particular string of words.

2.2.1 The Noisy Channel Model

Most probabilistic taggers are based on some variant of the n-class model, which can be seen as an instance of Shannon's noisy channel model based on Bayesian

inversion:

$$P(c_1, \ldots, c_k | w_1, \ldots, w_k) = \frac{P(w_1, \ldots, w_k | c_1, \ldots, c_k) P(c_1, \ldots, c_k)}{P(w_1, \ldots, w_k)}$$

In order to find the maximally probable part-of-speech sequence $c_1, \ldots, c_k$ for a given string of words $w_1, \ldots, w_k$, we only need to find that sequence which maximizes the product in the numerator of the right hand side (since the denominator is constant for a given word string). The first factor of this product is given by the *lexical model*:

$$\hat{P}(w_1, \ldots, w_k | c_1, \ldots, c_k) = \prod_{i=1}^{k} P(w_i | c_i)$$

In this model, every word is conditioned only on its own part-of-speech, an independence assumption which may seem unrealistic but which is necessary in order to get a tractable and trainable model. Some early systems (e. g., [DeRose 1988]) instead use the inverse probabilities, i. e., $P(c_i|w_i)$, which may be easier to estimate intuitively but which are not warranted by the noisy channel model and which appear to give worse performance ([Charniak *et al* 1993]).

The second factor is estimated by means of the *contextual model*:

$$\hat{P}(c_1, \ldots, c_k) = \prod_{i=1}^{k} P(c_i | c_{i-(n-1)}, \ldots, c_{i-1})$$

In this model, every part-of-speech is conditioned on the $n-1$ previous parts of speech. Depending on the value of n, we get different varieties of the n-class model, known as uniclass, biclass, triclass, etc. The two most common values of n are 2 and 3, and in this report we will restrict ourselves mainly to the biclass and triclass models.[4]

2.2.2 Hidden Markov Models

The n-class model can be implemented very efficiently as a Hidden Markov Model (HMM), consisting of:

- A set of states $S = \{s_1, \ldots, s_m\}$.
- A set of input symbols $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$.[5]
- A vector Π of initial state probabilities, where π_i is the probability that the model is in state s_i at time 1, i.e. $\pi_i = P(X_1 = s_i)$.
- A matrix A of state transition probabilities, where a_{ij} is the probability that the model is in state j at time k given that it is in state i at time $k-1$ ($k > 1$), i.e. $a_{ij} = P(X_k = s_j | X_{k-1} = s_i)$.

[4] We prefer to use the terms *n-class*, *biclass* and *triclass*, rather than the more common terms *n-gram*, *bigram* and *trigram*, to indicate that we are dealing with sequences of *word classes*, not *word forms*. Occasionally, we will also use the alternative terms *tag n-gram*, *tag bigram* and *tag trigram* to refer to sequences of n, two and three tags, respectively.

[5] In the standard description of Markov models (see, e.g., [Jelinek 1997]) symbols are usually described as *output* symbols being *generated* by the model. However, for our particular application, it is more natural to view them as *input* symbols being *read* by the model.

- A matrix B of input symbol probabilities, where b_{ij} is the probability that the model reads symbol j in state i, i.e. $b_{ij} = P(I_k = \sigma_j | X_k = s_i)$.

In practical implementations, this model is usually simplified by introducing a special start state s_0, in which the model is at time 0 with probability 1. The initial state probabilities can then be expressed as state transition probabilities from the special start state s_0, and the number of probability distributions is thereby reduced from three to two.

When using HMMs for part-of-speech tagging, input symbols correspond to word forms, while states correspond to tag sequences. The biclass model can be represented by a first-order HMM, where states represent single tags (sequences of length 1), while the triclass model requires a second-order HMM, where states represent tag pairs (sequences of length 2). More generally, an n-class tagging model corresponds to an $(n-1)$-order HMM, where each state represents a sequence of $n-1$ tags.

In this kind of implementation, the lexical probabilities of the n-class model are captured by input symbol probabilities, while contextual probabilities correspond to state transition probabilities. And the task of finding the most probable tag sequence for a given string of words is equivalent to finding the most probable path (state sequence) of the model for a particular input string. More precisely, given as input a sequence of words $w_1 \cdots w_k$, the task of finding that sequence of tags $c_1 \cdots c_k$ which maximizes the probability $P(c_1 \cdots c_k, w_1 \cdots w_k)$ is tantamount to finding the optimal path (state sequence) through the corresponding $(n-1)$-order Hidden Markov Model. Now, given a model with m states and an input sequence of length n, the number of possible paths through the model is m^n, which means that the running time of an exhaustive search grows exponentially with string length. Fortunately, there is a more efficient algorithm for finding the best path, known as the Viterbi algorithm ([Viterbi 1967]), which is linear in string length.

2.2.3 The Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm which exploits the fundamental property of Markov models, viz. that the probability for being in state s at time i only depends on the state of the model at time $i-1$. In general, this means that if we know the best way to get to every state at time $i-1$, we only need to consider extensions of these paths when constructing the set of candidate paths for time i. Let M be a model with states $\{s_0, s_1, \ldots, s_{m-1}\}$, with s_0 being the designated start state, let $w_1 \cdots w_n$ be the sequence of input symbols (words), let $\gamma_i(s_j)$ denote the best path to reach state s_j at time i while reading the input $w_1 \cdots w_i$, and let $p_i(s_j)$ denote the probability of this path (conditional on the input $w_1 \cdots w_i$). Then we have the following:

1. The best path to reach state s_0 at time 0, $\gamma_0(s_0)$, is s_0. The probability of this path, $p_0(s_0)$, is 1. [Note that $p_0(s_j) = 0$ for $j > 0$.]

2. The best path to reach state s_j $(0 < j < m)$ at time i $(i > 0)$, $\gamma_i(s_j)$, is the most probable of the paths $\gamma_{i-1}(s_k)s_j$ $(0 \leq k < m)$. The probability of this path, $p_i(s_j)$, is $p_{i-1}(s_k) \cdot P(s_j | s_k) \cdot P(w_i | s_j)$. [Note that $p_i(s_0) = 0$ for $i > 0$.]

3. The best path through the model while reading input $w_1, \ldots, w_n$ is the most probable of the paths $\gamma_n(s_j)$ $(0 \leq s_j < m)$. The probability of this path is $p_n(s_j)$.

Thus, for a given state s_j and an input sequence of length n, we only need to compute mn paths in order to find the best path to s_j. In order to find the best path overall we therefore need to compute m^2n paths, which means that the running time of the algorithm is quadratic in the number of states but linear in the length of the input. Since the number of states is constant for a given model, we may conclude that the time complexity of the Viterbi algorithm is $O(n)$, where n is the number of symbols in the input sequence.

2.2.4 Parameter Estimation

The major problem in constructing a probabilistic tagger — or any other probabilistic model for that matter — is to find good estimates for the model parameters. In the n-class model, there are two types of parameters that need to be estimated:

1. Lexical probabilities: $P(w|c)$
2. Contextual probabilities: $P(c_i|c_{i-(n-1)}, \ldots, c_{i-1})$

There are basically two methods that are used to estimate these parameters empirically from corpus data, depending on what kind of data is available for training. Both methods are based on the notion of Maximum Likelihood Estimation (MLE), which means that we try to choose those estimates that maximize the probability of the observed training data. If we have access to tagged training data, we can use relative frequencies to estimate probabilities:[6]

$$\hat{P}(w|c) = \frac{f(w,c)}{f(c)}$$

$$\hat{P}(c_i|c_{i-(n-1)}, \ldots, c_{i-1}) = \frac{f(c_{i-(n-1)}, \ldots, c_i)}{f(c_{i-(n-1)}, \ldots, c_{i-1})}$$

If we only have access to untagged data, the standard method is to start from some initial model and use the Baum-Welch algorithm for Hidden Markov Models ([Baum 1972]) to iteratively improve the estimates until we reach a local maximum.[7] Unfortunately, there is no guarantee that we ever reach a *global* maximum, and results are generally better if we can use tagged data for estimation ([Merialdo 1994]).

Regardless of which method we use to obtain a maximum likelihood estimation from our training data, we still have to face the ubiquitous problem of *sparse data*, which means that, for a lot of the events whose probability we want to estimate, we simply do not have enough data to get a reliable estimate. The most drastic case of this is events that do not occur at all in the training data, such as 'unknown words' in the context of part-of-speech tagging. If we assign

[6] The relative frequency $f_n(E)$ of an event E in a sample of n observations is always a maximum likelihood estimate of the probability $P(E)$; see, e. g., [Lindgren 1993].

[7] The Baum-Welch algorithm can be seen as a special case of the general technique known as Expectation-Maximization (EM); cf. [Dempster *et al* 1977].

these events zero probability (according to MLE), then any chain of independent events involving such an event will also be assigned probability zero, which is usually not very practical (unless we can be sure that the event in question is really impossible and not just infrequent). Therefore, we normally want to adjust our estimates in such a way that we can reserve some of the probability mass for events that we have not yet seen. This is what is known in the field as *smoothing*.

2.2.5 Smoothing

Before we turn to the various methods used for smoothing, let us note that the problem of sparse data affects the two models involved in statistical part-of-speech tagging rather differently. In the contextual model, we always know how many events we haven't seen. For example, given a part-of-speech system with n_{class} tags, we know that there are ${n_{class}}^n$ possible n-tuples. By contrast, the lexical model is open-ended, and it is usually very difficult to estimate how many words (or word-tag pairs) we haven't seen — unless we use a lexicon to stipulatively limit the class of words allowable in texts, a move which is often made when evaluating taggers, but which is usually completely unrealistic from a practical application point of view.

The methods used for smoothing can be divided into two broad categories. In the first category, which we may call *smoothing proper*, we find methods where the parameters of a single model are being adjusted to counter the effect of sparse data, usually by taking some probability mass from seen events and reserving it for unseen events. This category includes methods such as *additive smoothing* ([Lidstone 1920, Gale and Church 1990]), *Good-Turing estimation* ([Good 1953, Gale and Sampson 1995]), and various methods based on held-out data and cross-validation ([Jelinek and Mercer 1985, Jelinek 1997]).

In the second category, which we may call *combinatory smoothing*, we find methods for combining the estimates from several models. The most well-known methods in this category are probably *back-off smoothing* ([Katz 1987]) and *linear interpolation* ([Brown *et al* 1992]). In the following, we will restrict the discussion to those methods that will be implemented in the toolbox in section 3.

Additive Smoothing Perhaps the simplest of all smoothing methods is what is known as *additive* smoothing, and which consists in adding a constant k to all the frequencies (including the zero frequencies of unseen events) and then making a new maximum likelihood estimation. In other words, for each value x of a variable X, where x has the observed (absolute) frequency $f(x)$ in a sample of n observations, and X has a sample space of n_X possible values, we give the following estimate of $P(x)$:[8]

$$\hat{P}(x) = \frac{f(x) + k}{n + k\, n_X}$$

Depending on the value of k, this method has different names. For $k = 1$ it is known as Laplace's Law; for $k = 0.5$ it is known as Lidstone's Law or Expected Likelihood Estimation (ELE). The results from language modeling seem to indicate that additive smoothing is not a very good method, usually

[8] In these and following formulas, $P(x)$ is shorthand for $P(X = x)$ in the usual way.

overestimating the probability of unseen events ([Gale and Church 1994]). On the other hand, [Nivre 2000] obtained very good results when using the additive method for smoothing the contextual model in part-of-speech tagging.

Good-Turing Estimation Good-Turing estimation is a more sophisticated smoothing method, which uses expected frequencies of frequencies to reestimate the raw sample frequencies. More precisely, for any outcome with (absolute) frequency $f(x)$, we base our probability estimates on the reestimated frequency $f^*(x)$ derived by the following formula:

$$f^*(x) = (f+1)\frac{E(n_{f(x)+1})}{E(n_{f(x)})}$$

where n_f is the number of outcomes with frequency f and $E(X)$ is the expectation value of the variable X. In practice, there is no way of precisely calculating expected frequencies of frequencies, and different versions of Good-Turing estimation differ mainly in the way they estimate these values from the observed frequencies of frequencies (see, e. g., [Good 1953, Church and Gale 1991, Gale and Sampson 1995]). Good-Turing estimation has been shown to give good results for the lexical model in part-of-speech tagging ([Nivre 2000]).

Back-off Smoothing The basic idea in back-off smoothing is to use the basic MLE model for events which are frequent enough in the training data to have reliable estimates and to back off to a more general model for rare events, i. e., back off to a model where distinct outcomes in the first model are lumped together according to some equivalence relation. However, in order to get a correct probabilistic model, we must introduce a discounting factor for the first model probabilities (in order to reserve some probability mass for unseen or underestimated events) and a normalizing factor for the back-off probabilities (in order to make them comparable to the first model probabilities). For example, in language modeling, it is common practice to back off from a trigram to a bigram model (and from a bigram model to a unigram model if necessary):

$$\hat{P}(w_i|w_{i-2},w_{i-1}) = \begin{cases} (1-\delta_{w_{i-2},w_{i-1}})\frac{f(w_{i-2},w_{i-1},w_i)}{f(w_{i-2},w_{i-1})} & \text{if } f(w_{i-2},w_{i-1},w_i) > t \\ \alpha_{w_{i-2},w_{i-1}}\hat{P}(w_i|w_{i-1}) & \text{otherwise} \end{cases}$$

In this equation, t is the frequency threshold above which we keep the estimates of the original model, $\delta_{w_{i-2},w_{i-1}}$ is the discounting factor for the context w_{i-2}, w_{i-1}, and $\alpha_{w_{i-2},w_{i-1}}$ is the normalization factor for this context. Different ways of determining these factors give different versions of back-off smoothing. One common method is to use some version of Good-Turing estimation for this task (see, e. g., [Katz 1987]).

In part-of-speech tagging, back-off smoothing can be used in the contextual model, which is structurally isomorphic to the n-gram model in language modeling. Thus, triclass probabilities may be smoothed as follows:

$$\hat{P}(c_i|c_{i-2},c_{i-1}) = \begin{cases} (1-\delta_{c_{i-2},c_{i-1}})\frac{f(c_{i-2},c_{i-1},c_i)}{f(c_{i-2},c_{i-1})} & \text{if } f(c_{i-2},c_{i-1},c_i) > t \\ \alpha_{c_{i-2},c_{i-1}}\hat{P}(c_i|c_{i-1}) & \text{otherwise} \end{cases}$$

For the lexical model, however, there is generally no sensible model to back off to. It is true that we could use the plain word probability $P(w)$ to estimate

the lexical probability $P(w|c)$, but the problem is that in cases where w is an unknown word (or even a very rare one), estimates of the former probability are usually no better than estimates of the latter.

Linear Interpolation The idea behind linear interpolation is similar to that of backoff smoothing in that it tries to combine several models instead of improving a single model. But in this case we use an estimate which is a weighted sum of the estimates from all available models. Given n models, this requires that we define constants $\lambda_1, \ldots \lambda_n$ ($0 \leq \lambda_i \leq 1$, $\sum_{i=1}^{n} \lambda_i = 1$). Thus, for the triclass model, we define in the estimates in the following way:

$$\hat{P}_{int}(c_i|c_{i-2}, c_{i-1}) = \lambda_1 \hat{P}(c_i) + \lambda_2 \hat{P}(c_i|c_{i-1}) + \lambda_3 \hat{P}(c_i|c_{i-2}, c_{i-1})$$

where $\hat{P}(c_i)$, $\hat{P}(c_i|c_{i-1})$ and $\hat{P}(c_i|c_{i-2}, c_{i-1})$ are the maximum likelihood estimates for the uniclass, biclass and triclass model, respectively. The main problem in using linear interpolation is to find good values for the constants λ_i. One way of solving this problem is to use an HMM (with ϵ-transitions) to represent the interpolated model and train the model using the Baum-Welch algorithm ([Jelinek 1990]).

2.3 Logic Programming

In this section, we will give a very brief presentation of the logic programming framework used to implement the toolbox for probabilistic part-of-speech tagging. The presentation will be very selective, focusing only on features that will be important in the sequel. All the examples will be in Prolog syntax, since this is the language that has been used for the implementation, although most of what will be said will be true also for other logic programming languages. For a more comprehensive introduction to logic programming in general and Prolog in particular, the reader is referred to [Sterling and Shapiro 1986].

2.3.1 Deductive Databases

A pure logic program can be regarded as a deductive database, consisting of two types of clauses:

- Facts, or unit clauses, which state that some relation holds of a number of arguments. For example, the clause `fwc(can,vb,999)`, consisting of the three-place predicate `fwc/3`[9] with arguments `can`, `vb` and `999`, can be used to represent the fact the word form *can* with tag `vb` has a frequency of 999 (in some corpus).

- Rules, or conditional clauses, which define new relations in terms of other relations. For example, the rule `known(W) :- fwc(W,C,F)` can be used to define a known word as a word that has a frequency (for some tag). Formally, a rule consists of a *head*, which is an atomic formula, and a *body*, which is a conjunction of atomic formulas, separated by the operator `:-`. Variables (such as `W`, `C`, `F`) are always universally quantified with the (implicit) quantifiers taking scope over the entire clause.[10]

[9] The notation `p/n` is standardly used to indicate that `p` is a predicate with arity n.

[10] In Prolog, variables always begin with an uppercase letter or the underscore character `_`.

The real power of logic programming lies in the fact that we can use automated theorem proving to deduce new facts from the clauses in the deductive database. For example, consider the following three-clause database:

```
fwc(can,vb,999).
fwc(the,dt,9999).
known(W) :- fwc(W,C,F).
```

Now, suppose that we want to know whether *can* is a known word or not. Then we can use the theorem prover to prove the following goal:

```
known(can).
```

Furthermore, if we want to know all the words that are known, we can formulate the following (existentially quantified) goal:

```
known(W)
```

and let the theorem prover *backtrack* to find all possible instantiations of the variable `W` for which the goal can be proven (in this case, `W=can` and `W=the`).

2.3.2 Cuts and Negation

Most implementations of logic programming languages offer capabilities that go beyond the power of pure logic programs. One example is the infamous *cut* in Prolog, which is a goal that succeeds and prunes all alternative solutions to a given goal. For example, consider the following simple Prolog program:

```
known(W) :- fwc(W,C,F).
known(W) :- lex(W,C).
```

This program defines a word as known if it either occurs with some tag and some frequency in our corpus (`fwc(W,C,F)`) or is listed with a particular tag in some lexical database (`lex(W,C)`). Now, suppose that many words both occur in the corpus and are listed in the lexical database. Then we can still use backtracking to find all the known words, but this will be inefficient since in many cases the same goal will be proven twice. One way of rectifying this situation is of course to make sure that no word occurs with both the predicates `fwc/3` and `lex/2`. Another way is to introduce a cut (`!`) as follows:

```
known(W) :- fwc(W,C,F), !.
known(W) :- lex(W,C).
```

Now, if the goal `known(W)` can be proven using the first rule, i.e. if we reach the cut, no further backtracking will be allowed and the second rule will never be tried. However, if the first rule fails, then the second rule will be tried as before.

Another use of the cut, in combination with the special predicate `fail` (which always fails), is to define a limited form of negation known as *negation as failure* and denoted by the symbol `\+`:

```
\+ P :- P, !, fail.
\+ P.
```

In order to prove `\+ P` ('not P'), we first try to prove `P`. If this succeeds, we fail (because of the `fail` predicate) and no further clauses are tried (because of the cut). If we fail to prove `P`, we try the second clause, which always succeeds. In other words, `\+ P` effectively means that `P` is not provable, which is equivalent to the negation of `P` under the closed world assumption (the assumption that all relevant facts are in the database).

Armed with negation as failure, we can now define an unknown word in the obvious way as a word that is not known (or cannot be proven to be known):

```
unknown(W) :- \+ known(W).
```

2.3.3 Second-Order Programming

In many cases, it is very useful to have predicates that produce sets as solutions. In particular, we are often interested in finding the set of all instances of a term that satisfy a particular goal. Most Prolog implementations offer at least two built-in predicates for finding all solutions to a particular goal:

- `bagof(`*`Term`*`,`*`Goal`*`,`*`Instances`*`)` is true if *`Instances`* is a list of all instances of *`Term`* for which *`Goal`* is true. Multiple identical solutions, corresponding to different proofs, are retained.
- `setof(`*`Term`*`,`*`Goal`*`,`*`Instances`*`)` is a refinement of `bagof/3`, where the list *`Instances`* is sorted and duplicate elements are removed.

In most implementations, both `bagof/3` and `setof/3` fail when there are no solutions to *`Goal`*. A third alternative, `findall/3`, always succeeds exactly once, and instantiates *`Instances`* to the empty list (`[]`) when no solutions are found.

Another kind of set-valued predicate is `length(`*`List`*`,`*`N`*`)`, which is true if *`List`* is a list containing *`N`* elements. For example, in order to find the number of known words in our database, we prove the following conjunctive goal:

```
setof(W,known(W),Ws), length(Ws,N)
```

If instead we want to count the number of known word-tag pairs (taking into account the fact that a single word form may occur with more than one tag), we instead use `bagof/3`:

```
bagof(W,known(W),Ws), length(Ws,N)
```

2.3.4 Modules and Interfaces

As stated in the introduction, we think of a toolbox as a collection of small components, implemented as logic programs, that can be combined to build larger programs, in our case probabilistic part-of-speech taggers. Each program in the toolbox will provide a definition of one or more logical predicates that are made available to the rest of the system. In this sense, each program can be regarded as a *module*, with its main predicate providing an *interface* to other modules.

However, these modules mainly exist on the conceptual level. When different components are compiled together into a complete tagging system, the whole

system will behave as a single logic program, and the modules can only be distinguished as subsets of program clauses.[11] Nevertheless, we will frequently use the module-interface metaphor in the sequel, when we think that it facilitates understanding.

[11] In fact, many logic programming environments support the use of modules in the more strict sense, but this is not an option that we will exploit here.

Chapter 3

The Basic Toolbox

Conceptually, a probabilistic tagger consists of four different modules:

Lexical module This modules computes lexical probabilities, that is, probabilities of words conditioned on tags, normally of the form $P(w|c)$.

Contextual module This module computes contextual probabilities, that is, probabilities of tags conditioned on neighboring tags, normally of the form $P(c_i|c_{i-(n-1)}, \ldots, c_{i-1})$ (where n is 2 or 3).

Search module The task of the search module is to find the most probable tag sequence for a given word sequence, which normally amounts to finding the optimum path (state sequence) through an $(n-1)$-order Hidden Markov Model. The parameters of the model are given by the lexical and contextual modules, respectively.

Input/Output module The Input/Output module is responsible for tokenizing and segmenting the input to the search module, and for printing the output in whatever format is required by the application.

The data flow and dependencies between modules is illustrated in Figure 3.1. Input text is first pre-processed and segmented by the I/O module and passed to the search module in chunks of suitable size. The search module constructs the tag sequence corresponding to a particular string of words, using probabilities from the lexical and contextual modules, then passes the analysis back to the I/O module which performs the necessary post-processing and outputs the tagged text in whatever format is required.

In the following sections, we will present tools for building each of the four typical modules in a probabilistic tagger, although the I/O module will be covered very briefly. In discussing the lexical and contextual modules, I will assume that we have available frequency data (usually derived from a pre-tagged corpus) that can be used to estimate the parameters of the different probabilistic models handled by these modules. More precisely, we will assume as given the following four Prolog databases:

fwc.pl The frequency of the word w occurring with tag c, i.e., $f(w, c)$, expressed in clauses of the form `fwc(w,c,n)`, where $n = f(w, c)$.

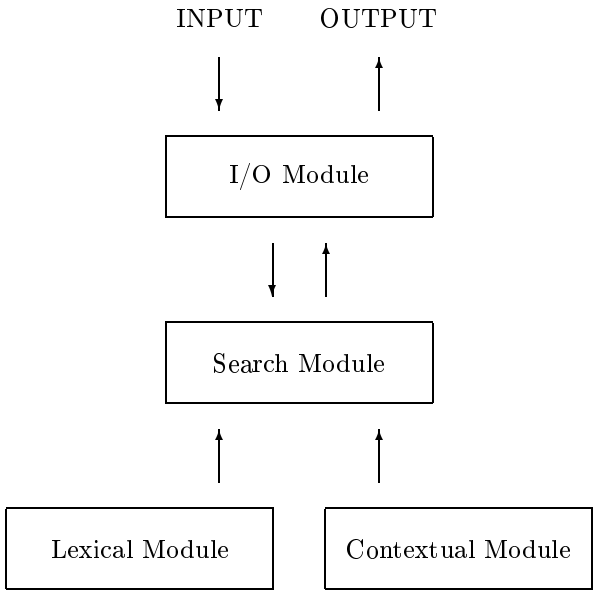

Figure 3.1: Modules and dependencies in a probabilistic tagger

fc.pl The frequency of the tag c, i.e., $f(c)$, expressed in clauses of the form `fc(c,n)`, where $n = f(c)$.

fcc.pl The frequency of the tag bigram (c_i, c_j), i.e., $f(c_i, c_j)$, expressed in clauses of the form `fcc(`c_i`,`c_j`,n)`, where $n = f(c_i, c_j)$.

fccc.pl The frequency of the tag trigram (c_i, c_j, c_k), i.e., $f(c_i, c_j, c_k)$, expressed in clauses of the form `fccc(`c_i`,`c_j`,`c_k`,n)`, where $n = f(c_i, c_j, c_k)$.

For an example, see appendix B which shows a small toy corpus, consisting of two sentences, and lists the corresponding frequency databases. We also need information about the following parameters:

n_{token} The number of tokens in the training corpus, expressed in a clause of the form `tokens(`n_{token}`)`.

n_{type} The number of types in the training corpus, expressed in a clause of the form `types(`n_{type}`)`.

n_{class} The number of classes (tags) used in the training corpus, expressed in a clause of the form `classes(`n_{class}`)`.

3.1 Tools for the Lexical Module

The task of the lexical module is to provide estimates of lexical probabilities, which are conditional probabilities of the form $P(w|c)$, where w is a word form

and c is a tag, based on the frequencies $f(w,c)$ and $f(c)$.[1] These estimates are normally based on the following maximum likelihood estimation (cf. section 2.2.4):

$$\hat{P}(w|c) = \frac{f(w,c)}{f(c)}$$

However, the estimates derived in this way are known to be problematic for at least three reasons:

1. Every unknown word w (i.e., a word form not occurring in the training data) has $P(w|c) = 0$ for every tag c, which is usually not desirable.

2. Every known word w has $P(w|c) = 0$ for every tag c with which it does not occur in the training data, which may or may not be desirable depending on the completeness of the training data.

3. Estimates for low-frequency items are unreliable.

These problems are all instances of the notorious 'sparse data problem', which means that, regardless of the size of the training corpus, there will always be some events that are too infrequent to get reliable estimates. The solution lies in *smoothing* the estimates, usually by applying the Robin Hood principle of taking some probability mass from observed events (the rich) and giving it to unobserved events (the poor). In the rest of this section, we will therefore discuss the implementation in Prolog of four different methods of estimation, three of which involves smoothing in this sense:

1. Maximum likelihood estimation (MLE)

2. Uniform models

3. Additive smoothing

4. Good-Turing estimation

In each case, the implementation consists in defining the predicate `pw_c/3`, which is true of a word form w, a tag c and a probability p iff $P(w|c) = p$ according to the model in question (and the given frequency data). This predicate provides the interface between the search module and the lexical module, in the sense that the search module will typically call the lexical module with a partially instantiated goal of the form `pw_c(w,C,P)`, where `C` and `P` are variables, and the lexical module will return all permissible tags with corresponding probabilities.

However, before we turn to the different methods of estimation, we need to discuss two more basic decisions. The first is whether to allow known words to occur with unknown tags, i.e., whether a known word w may have $P(w,c) > 0$ for any tag c with which it does not occur in the training data. Despite the fact that most training corpora are incomplete, I will assume in the following

[1] Strictly speaking, the probability required by an n-order Markov model is $P(w|s)$, where s is a state of the model. For the first-order model, there is a one-to-one correspondence between tags and states, but for higher-order models, states correspond to ordered sets of tags. However, in practice, lexical probabilities are seldom conditioned on more than one tag, because of the sparse data problem, so we will ignore this complication for the moment (but cf. sections 3.3.3 and 4.2.4).

that this is not allowed. The reason is that problems caused by known words occurring with unknown tags are relatively infrequent, as compared to the other two problems, and attempts to cope with them therefore tend to result in a lower overall accuracy.

The second decision concerns which tags to allow for unknown words, i.e. which parts-of-speech to regard as *open*. Given a training corpus of reasonable size, we can usually expect traditional closed classes such as conjunctions, pronouns and prepositions to be fairly complete, whereas open classes such as nouns, verbs, and adjectives will always be incomplete, but where to draw the line for a given data set is not a trivial problem.[2] In the following, we will simply assume that there is a predicate `open/1`, which is true of every part-of-speech c that is treated as open in the sense that it may be assigned (with non-zero probability) to unknown words.

3.1.1 Maximum Likelihood Estimation

The following definition of `pw_c/3` gives us the maximum likelihood model (MLE):

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     fc(C,F2),
     P is F1/F2.

pw_c(W,C,0) :-
     \+ fwc(W,_,_),
     open(C).
```

For known words, the lexical probability given a particular class is simply the ratio between the word-tag frequency and the tag frequency. For unknown words, the lexical probability given a particular *open* class is zero. In the second clause, we have used negation as failure to make sure that the two cases are mutually exclusive so that there is at most one probability defined for each word-tag pair. This is a technique that will be used consistently in the definition of probability estimates both for the lexical and the contextual module. An alternative (and more efficient) solution would have been to use a cut as follows:

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     !,
     fc(C,F2),
     P is F1/F2.

pw_c(W,C,0) :-
     open(C).
```

However, since optimization is not an issue at this point, we prefer to use the more explicit and logically transparent definition above. Yet a third alternative would have been to use the predicate `unknown/1` defined in section 2.3.2. The second clause would then become:

[2] See [Nivre 2000] for an attempt to use statistical measures to settle this question.

```
pw_c(W,C,0) :-
     unknown(W),
     open(C).
```

Although this is perhaps a more elegant solution, we will stick to the original formulation with negation in what follows.

3.1.2 Uniform Models

A simple yet useful method for avoiding the zero probabilities associated with unknown words in the basic MLE model is to treat all unknown words as occurrences of a single unknown word type w_u and let $P(w_u|c)$ be constant for all open parts-of-speech. Such a model is uniform in the sense that every open part-of-speech has the same probability to occur with the unknown word, which in practice means that we leave the decision to the contextual model. In the following implementation, we use $1/n_{token}$ as the uniform lexical probability for unknown words:

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     fc(C,F2),
     \+ open(C),
     P is F1/F2.

pw_c(W,C,P) :-
     fwc(W,C,F1),
     fc(C,F2),
     open(C),
     tokens(N),
     P is F1/(F2+(F2/N)).

pw_c(W,C,P) :-
     \+ fwc(W,_,_),
     open(C),
     tokens(N),
     P is 1/N.
```

In order to get a correct probabilistic model, the probability assigned to known words in open classes has to be adjusted to allow for one occurrence of the unknown word in each class. For closed classes, however, we use the basic MLE model. Therefore, the program contains three clauses — one for closed classes, one for known words in open classes, and one for unknown words in open classes — which are made mutually exclusive using negative conditions.

3.1.3 Additive Smoothing

Additive smoothing consists in adding a constant k to all the frequencies (including the zero frequencies of unknown words) and then making a new maximum likelihood estimation (cf. section 2.2.5). The following program implements additive smoothing of the lexical model (given that the additive konstant k is defined by a clause of the form `lex_add(`k`)`):

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     fc(C,F2),
     lex_add(K),
     types(N),
     P is (F1+K)/(F2+(K*(N+1))).

pw_c(W,C,P) :-
     \+ fwc(W,_,_),
     open(C),
     fc(C,F),
     lex_add(K),
     types(N),
     P is K/(F+(K*(N+1))).
```

Note that, as before, known words are not allowed to have unknown tags, and unknown words can only be assigned to open classes, even though the additive model as such permits any word to have any tag.

The additive method is conceptually simple and easy to implement but have been shown to give inferior results when applied to the lexical model ([Nivre 2000]). Since the uniform model is equally simple (but usually gives better performance), additive smoothing of the lexical model is little used in practice and included here mainly for reference.

3.1.4 Good-Turing Estimation

As explained in chapter 2, Good-Turing estimation is a more sophisticated smoothing method, which uses expected frequencies of frequencies to reestimate the raw sample frequencies (cf. section 2.2.5). The following implementation presupposes that frequencies have been reestimated separately for different parts-of-speech and stored in a database with clauses of the form `gtfwc(c,`f_1`,`f_2`)`, where f_2 is the reestimated frequency corresponding to frequency f_1 for tag c:

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     \+ open(C),
     fc(C,F2),
     P is F1/F2.

pw_c(W,C,P) :-
     fwc(W,C,F1),
     open(C),
     gtfwc(C,F1,F2),
     fc(C,F3),
     P is F2/F3.

pw_c(W,C,P) :-
     \+ fwc(W,_,_),
     open(C),
     gtfwc(C,0,F1),
     fc(C,F2),
```

```
P is F1/F2.
```

As in the case of the uniform model, there are three clauses in the definition, corresponding to closed classes, known words in open classes, and unknown words in open classes, respectively.

Experiments have shown that Good-Turing smoothing gives particularly good results for the lexical model, where it is arguably the method of choice ([Nivre 2000]). However, it does require the extra work of calculating reestimated frequencies for all open classes.[3]

3.1.5 Unknown Words

The major problem that affects the lexical module is the problem of unknown words. All the methods discussed in this section have in common that they treat all unknown words as occurrences of a single unknown word type w_u and do not exploit any information other than the distribution of known words across different parts-of-speech. In practice, tagging accuracy can be increased considerably by considering formal properties of unknown words, in particular word endings and capitalization. Sometimes this information is integrated in the probabilistic model defined in the lexical module, sometimes it is implemented in the form of heuristic rules applied in a separate post-processing step. We will discuss both kinds of methods in section 4.2, where we consider ways of improving tagging accuracy. For the time being, we will content ourselves with the 'pure' statistical methods treated in this section.

3.2 Tools for the Contextual Module

The task of the contextual module is to provide estimates of contextual probabilities, which are conditional probabilities of the form $P(c_i|c_{i-1})$ (biclass) and $P(c_i|c_{i-2}, c_{i-1})$ (triclass), where c_{i-2}, c_{i-1} and c_i are tags, based on the frequencies $f(c_i)$, $f(c_{i-1})$, $f(c_{i-1}, c_i)$, $f(c_{i-2}, c_{i-1})$, $f(c_{i-2}, c_{i-1}, c_i)$. These estimates are normally based on the following maximum likelihood estimations (cf. section 2.2.4):

$$\hat{P}(c_i|c_{i-1}) = \frac{f(c_{i-1}, c_i)}{f(c_{i-1})}$$

$$\hat{P}(c_i|c_{i-2}, c_{i-1}) = \frac{f(c_{i-2}, c_{i-1}, c_i)}{f(c_{i-2}, c_{i-1})}$$

As in the case of lexical probabilites, the pure maximum likelihood estimates are likely to be problematic because of data sparseness, although the problem is less severe for the contextual models (especially the biclass model) because of the smaller event space ([Nivre 2000]). Therefore, we will discuss the implementation in Prolog of five different methods of estimation, four of which involves smoothing of the basic MLE model:

1. Maximum likelihood estimation (MLE)
2. Additive smoothing

[3] A program for calculating the reestimated frequencies using the simple Good-Turing method ([Gale and Sampson 1995]), written by Dan Melamed, is available by ftp from: `ftp://ftp.cis.upenn.edu/pub/melamed/tools/Good-Turing_smoothing/`.

3. Good-Turing estimation

4. Backoff smoothing

5. Linear interpolation

In each case, the Prolog implementation consists in defining the two predicates `pc_c/3` and `pc_cc/4`. The former is true of tags c_i, c_{i-1} and probability p iff $P(c_i|c_{i-1}) = p$; the latter is true of tags c_i, c_{i-2}, c_{i-1} and probability p iff $P(c_i|c_{i-2}, c_{i-1}) = p$. That is, given the frequency data and the model chosen, we have:

$$\texttt{pc\_c}(c_i, c_{i-1}, p) \Leftrightarrow P(c_i|c_{i-1}) = p$$
$$\texttt{pc\_cc}(c_i, c_{i-2}, c_{i-1}, p) \Leftrightarrow P(c_i|c_{i-2}, c_{i-1}) = p$$

These predicates provide the interface between the search module and the contextual module, and the search module will typically call the contextual module with goals of the form $\texttt{pc\_c}(c_i, c_{i-1}, \texttt{P})$ and $\texttt{pc\_cc}(c_i, c_{i-2}, c_{i-1}, \texttt{P})$, where `P` is a variable which will be instantiated to the appropriate probability. (For the sake of completeness, we will also define the predicate `pc/2`, where $\texttt{pc}(c, p)$ is true iff $P(c) = p$.)

Before we turn to the different methods of estimation for contextual probabilities, we need to sort out a small technical detail. In order to simplify the implementation of the contextual module, it is common practice to introduce dummy tags for the (non-existent) words preceding the first word in the sequence to be tagged. In this way, the contribution of the first word(s) to the probability of the tag sequence can be treated in terms of contextual probabilities as well, thus avoiding the need for special cases handling initial probabilities.[4] In what follows, we will assume that the dummy tag is `start`, and that frequencies involving this tag are defined in such a way that the following equations hold:

$$\frac{f(\texttt{start}, c)}{f(\texttt{start})} = \frac{f(\texttt{start}, \texttt{start}, c)}{f(\texttt{start}, \texttt{start})} = \hat{P}_{initial}(c)$$

$$\frac{f(c_i, c_j)}{f(c_i)} = \frac{f(\texttt{start}, c_i, c_j)}{f(\texttt{start}, c_i)} = \hat{P}(c_j|c_i)$$

The way in which these equations will be satisfied depends on choices having to do with the way in which the training corpus is organized. A simple yet effective solution is to use the following values:

$$f(\texttt{start}) = f(\texttt{start}, \texttt{start}) = n_{token}$$

$$f(\texttt{start}, c) = f(\texttt{start}, \texttt{start}, c) = f(c)$$

$$f(\texttt{start}, c_i, c_j) = f(c_i, c_j)$$

Given one of the estimation methods discussed below, the resulting model simply equates initial probabilities with occurrence probabilities anywhere in a sequence. It is important to note, however, that this does not necessarily mean that we equate the probability of occurring at the start of a *sentence* (or any other linguistic unit) with the probability of occurring in any position, since the

[4] This corresponds to the introduction of a special start state s_0 in the HMM (cf. section 2.2.2).

former position can be recognized as the position immediately following a token tagged as a major delimiter. Only if we feed the text to the tagger sentence by sentence will the notions 'start of sequence' and 'start of sentence' coincide (cf. section 3.4). In the latter case, we may want to define the frequencies involving the dummy tag `start` in terms of occurrences after a major delimiter in the training corpus. In any case, we do not want to prescribe a particular solution once and for all, but instead leave it up to the individual researcher/developer using the toolbox to make an informed decision on this point.

3.2.1 Maximum Likelihood Estimation

The definition of maximum likelihood estimates in terms of relative frequencies is straightforward:

```
pc(C,P) :-
     fc(C,F),
     tokens(N),
     P is F/N.

pc_c(C2,C1,P) :-
     fcc(C1,C2,F1),
     fc(C1,F2),
     P is F1/F2.

pc_c(C2,C1,0) :-
     \+ fcc(C1,C2,_).

pc_cc(C3,C1,C2,P) :-
     fccc(C1,C2,C3,F1),
     fcc(C1,C2,F2),
     P is F1/F2.

pc_cc(C3,C1,C2,0) :-
     \+ fccc(C1,C2,C3,_).
```

The special cases involving negation as failure are needed in order to assign zero probabilities to non-occurring tag combinations. These cases can in principle be eliminated by making sure that the frequency database also contains all zero frequencies. However, this move is inadvisable for two reasons. First, for large tagsets with sparse data the space requirements may increase dramatically. For example, given 100 tags, there are 1 000 000 possible tag trigrams, only a fraction of which will normally be found in the training corpus. Secondly, we may end up dividing by zero when computing triclass probabilities in cases where not even the context biclass occurs in the training corpus.

3.2.2 Additive Smoothing

The implementation of additive smoothing is parallel to the implementation for the lexical module discussed in section 3.1.3:

```
pc(C,P) :-
```

```
        fc(C,F),
        tokens(N),
        P is F/N.

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        fc(C1,F2),
        con_add(K),
        classes(N),
        P is (F1+K)/(F2+(K*N)).

pc_c(C2,C1,P) :-
        \+ fcc(C1,C2,_),
        fc(C1,F),
        con_add(K),
        classes(N),
        P is K/(F+(K*N)).

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        fcc(C1,C2,F2),
        con_add(K),
        classes(N),
        P is (F1+K)/(F2+(K*N*N)).

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        fcc(C1,C2,F),
        con_add(K),
        classes(N),
        P is K/(F+(K*N*N)).

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        \+ fcc(C1,C2,_),
        con_add(K),
        classes(N),
        P is K/(K+(K*N)).
```

We use a predicate `con_add/1` to define the value of the additive constant (analogous to `lex_add/1` for the lexical module). Note that we need three special cases to handle unknown tag combinations, since we need to distinguish the case where the triclass is unknown but the context biclass is known (in which case the goal `fcc(C1,C2,F)` succeeds) from the case where both the triclass and the biclass are unknown (in which case it fails). Note also that we have kept the MLE model for uniclass probabilities, since we assume that all tags have non-zero frequency. This will be assumed for all estimations methods that follow as well.

3.2.3 Good-Turing Estimation

The implementation of Good-Turing estimation is also parallel to the implementation for the lexical module (cf. section 3.1.4):

```
pc(C,P) :-
     fc(C,F),
     tokens(N),
     P is F/N.

pc_c(C2,C1,P) :-
     fcc(C1,C2,F1),
     gtfcc(F1,F2),
     fc(C1,F3),
     P is F2/F3.

pc_c(C2,C1,P) :-
     \+ fcc(C1,C2,_),
     gtfcc(0,F1),
     fc(C1,F2),
     P is F1/F2.

pc_cc(C3,C1,C2,P) :-
     fccc(C1,C2,C3,F1),
     gtfccc(F1,F2),
     fcc(C1,C2,F3),
     gtfcc(F3,F4),
     P is F2/F4.

pc_cc(C3,C1,C2,P) :-
     \+ fccc(C1,C2,C3,_),
     gtfccc(0,F1),
     fcc(C1,C2,F2),
     gtfcc(F2,F3),
     P is F1/F3.

pc_cc(C3,C1,C2,P) :-
     \+ fccc(C1,C2,C3,_),
     gtfccc(0,F1),
     \+ fcc(C1,C2,_),
     gtfcc(0,F2),
     P is F1/F2.
```

The predicates `gtfcc/2` and `gtfccc/2` define the reestimated frequency corresponding to a particular observed frequency for the biclass and triclass model, respectively. As in the case of additive smoothing, there are three special cases for unknown tag combinations (one for the biclass model and two for the triclass model).

3.2.4 Backoff Smoothing

The basic idea in backoff smoothing is to use the underlying MLE model as long as the observed frequency f of the n-class is above a certain threshold t, and to retreat to the more robust $n-1$-class when $f \leq t$. However, in order to get a correct probability distribution, we have to introduce a discounting factor d ($0 < d < 1$) that determines how large a proportion of the entire probability mass is reserved for the proper n-class estimates, with the remaining proportion being reserved for the back-off estimates derived from the $n-1$-class model. This discounting factor can furthermore be made sensitive to the frequency distribution for a particular context using, for example, Good-Turing estimation to determine how large proportion of the probability mass to reserve for unseen events (cf. section 2.2.5).

The following logic program implements a special case of backoff smoothing with a single constant discounting factor d for all contexts. We assume that the frequency threshold t and the discounting factor d are defined by the predicates `threshold/1` and `discount/1`, respectively, but the actual choice of values for these constants are left to the individual researcher/developer:

```
pc(C,P) :-
     fc(C,F),
     tokens(N),
     P is F/N.

pc_c(C2,C1,P) :-
     fcc(C1,C2,F1),
     threshold(T),
     F1 > T,
     fc(C1,F2),
     discount(D),
     P is (1-D)*F1/F2.

pc_c(C2,C1,P) :-
     fcc(C1,C2,F1),
     threshold(T),
     \+ F1 > T,
     pc(C2,P1),
     discount(D),
     P is D*P1.

pc_c(C2,C1,P) :-
     \+ fcc(C1,C2,_),
     pc(C2,P1),
     discount(D),
     P is D*P1.

pc_cc(C3,C1,C2,P) :-
     fccc(C1,C2,C3,F1),
     threshold(T),
     F1 > T,
     fcc(C1,C2,F2),
```

```
     discount(D),
     P is (1-D)*F1/F2.

pc_cc(C3,C1,C2,P) :-
     fccc(C1,C2,C3,F1),
     threshold(T),
     \+ F1 > T,
     pc_c(C3,C2,P1),
     discount(D),
     P is D*P1.

pc_cc(C3,C1,C2,P) :-
     \+ fccc(C1,C2,C3,_),
     fcc(C1,C2,_),
     pc_c(C3,C2,P1),
     discount(D),
     P is D*P1.

pc_cc(C3,C1,C2,P) :-
     \+ fccc(C1,C2,C3,_),
     \+ fcc(C1,C2,_),
     pc_c(C3,C2,P).
```

Note that it is again necessary to have special cases for zero frequencies, even though in principle these should be covered by the case $f \leq t$, the reason being that the goals `fcc(C1,C2,F1)` and `fccc(C1,C2,C3,F1)` will fail in these cases. Moreover, in the very special case where the context is unknown (which can only happen for the triclass model), we use the $n-1$-class estimate directly, without any discounting factor, since there are no estimates at all available from the n-class model.

3.2.5 Linear Interpolation

The idea behind linear interpolation is similar to that of backoff smoothing in that it tries to combine several models instead of improving a single model, but instead of skipping from one model to the other, we always use an estimate which is a weighted sum of the estimates from all available models. This requires that we define constants $\lambda_1, \ldots \lambda_n$ ($0 \leq \lambda_i \leq 1$, $\sum_{i=1}^{n} \lambda_i = 1$). In the following implementation of linear interpolation for the contextual model, we assume that these constants are given by the predicates `bi1/1` and `bi2/1` for the biclass model and `tri1/1`, `tri2/1` and `tri3/1` for the triclass model:

```
pc(C,P) :-
     fc(C,F),
     tokens(N),
     P is F/N.

pc_c(C2,C1,P) :-
     fcc(C1,C2,F1),
     fc(C1,F2),
     pc(C2,P1),
```

```
      bi1(K1),
      bi2(K2),
      P is K1*P1+K2*F1/F2.

pc_c(C2,C1,P) :-
      \+ fcc(C1,C2,_),
      pc(C2,P1),
      bi1(K1),
      P is K1*P1.

pc_cc(C3,C1,C2,P) :-
      fccc(C1,C2,C3,F1),
      fcc(C1,C2,F2),
      pc(C3,P1),
      pc_c(C3,C2,P2),
      tri1(K1),
      tri2(K2),
      tri3(K3),
      P is K1*P1+K2*P2+K3*F1/F2.

pc_cc(C3,C1,C2,P) :-
      \+ fccc(C1,C2,C3,_),
      pc(C3,P1),
      pc_c(C3,C2,P2),
      tri1(K1),
      tri2(K2),
      P is K1*P1+K2*P2.
```

3.3 Search Tools

The task of the search module is to find the most probable part-of-speech sequence $c_1 \cdots c_n$ for a given word sequence $w_1 \cdots w_n$, given the probabilistic model jointly defined by the lexical and contextual modules.

The search module is interfaced via the predicate `most_probable_path/2`, which is true of word sequence $w_1 \cdots w_n$ and a pair $(p, c_1 \cdots c_n)$, consisting of probability p and tag sequence $c_1 \cdots c_n$ iff $c_1 \cdots c_n = argmax_{C_{1,n}} P(w_1 \cdots w_n, C_{1,n})$ and $p = P(w_1 \cdots w_n, c_1 \cdots c_n)$.

Both word sequences and tag sequences are represented in Prolog as lists of atoms, and the tag sequence and probability occur as arguments of an infix operator `-`. For example, the fact that `dt jj nn` is the most probable tag sequence for the word sequence `the red car`, with probability 0.001 is represented as follows:

```
most_probable_path([the,red,car],0.001-[dt,jj,nn])
```

The search module accesses the lexical and contextual modules via the predicates `pw_c/3`, `pc/2`, `pc_c/3` and `pc_cc/4` (cf. sections 3.1–3.2).

As explained in section 2.2.3, the standard HMM algorithm for finding the optimal path for a given input is the Viterbi algorithm ([Viterbi 1967]). In this section, we will consider three different implementations of the Viterbi algorithm

in Prolog. We will begin with an implementation for the biclass tagging model, which is due to [Lager 1995]. Next, we will look at a simple modification of the first program to fit the triclass tagging model as usually implemented.[5] Finally, we will develop an implementation which handles the full second-order Markov model, in which lexical probabilities may be conditioned on pairs of tags instead of only single tags.

3.3.1 Biclass Viterbi

The biclass model (cf. section 2.2) corresponds to a first-order Markov model, where each state represents one tag. In the following implementation of the Viterbi algorithm, states are represented directly by tags, paths by lists of tags/states, and input sequences by lists of words.

The top-level predicate is `most_probable_path/2`, which is defined in the following way:

```
most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start]],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start|Path]).
```

This clause says that the most probable path for the list of words `Words` is the most probable of the paths that can can be constructed for the input `Words` starting from the path `[start]` with probability `1` (where `start` represents the generic start state). The heart of the program is in the predicate `paths/3`, which will be discussed in detail presently. The predicate `most_probable/2` simply runs through the list of Probability-Path pairs and selects the pair with highest probability. And since the path is constructed by entering tags/states at the head of the path-list as we go through the input string, we also need to reverse this list (and get rid of the start state) in order to get the correct sequence of tags; hence the call to `reverse/2`.

The predicate `paths/3` is a recursive program that computes $\gamma_n(s_j)$ and $p_n(s_j)$ for every state s_j (cf. section 2.2.3):

```
paths([],PPs,PPs).

paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_c(Word,Class2,LexP),
            findall(P1-[Class2,Class1|RestPath],
                    (member(P0-[Class1|RestPath],PPs0),
                    pc_c(Class2,Class1,ConP),
                    P1 is LexP*ConP*P0),
                PPs1),
            most_probable(PPs1,PP)),
        PPs2),
    paths(Words,PPs2,PPs).
```

The basis clause says that if the input list is empty, then whatever paths we have constructed so far are the best paths for the given input. (Given the empty

[5] Thanks to James Cussens for finding and fixing a bug in this program.

input list to start with, `paths/2` returns the path `[start]` with probability `1` as expected.)

If the input list is not empty, then, for every possible tag (`Class2`) of the first word (`Word`), we extend the paths constructed so far (members of `PPs0`) and compute their probability by multiplying the path probability (`P0`) with the lexical probability `LexP` of `Word` given `Class2` and the contextual probability `ConP` of `Class2` given the previous tag `Class1`. We then find the most probable (extended) path for each possible tag `Class2`, store these with their probabilities in a list (`PPs2`) and recursively call `paths/2` with these paths and the rest of the input list (`Words`).

The implementation uses two nested calls to `findall/3`, which gives a very compact and elegant implementation of the Viterbi algorithm. The innermost call finds all the possible path extensions for a given tag and computes their probabilities; the outermost call finds all the possible tags for a given word and selects, for each tag, the most probable of the path extensions.

3.3.2 Triclass Viterbi

When moving from the biclass model to the triclass model, we move from a first-order to a second-order Markov model. This means that each state in the model represents a pair of tags and that, if we continue to represent paths by lists of tags, there will no longer be a one-to-one mapping from a path element to a state (only to 'half a state' as it were). Thus, when we consider transition probabilities, we have to take into account not only the last element of a candidate path but the last *two* elements. Here is a minimal modification of the implementation of `paths/3` from the preceding section that solves this problem:

```
paths([],MPPs,MPPs).

paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_c(Word,Class3,LexP),
            pc(Class2,_),
            findall(P1-[Class3,Class2,Class1|RestPath],
                    (member(P0-[Class2,Class1|RestPath],PPs0),
                    pc_cc(Class3,Class1,Class2,ConP),
                    P1 is LexP*ConP*P0),
                PPs1),
            most_probable(PPs1,PP)),
        PPs2),
    paths(Words,PPs2,PPs).
```

This program differs from its predecessor in three places. First of all, the tag lists in `PPs0` are specified with respect to their first *two* members — `Class2` and `Class1` — instead of just one. In the same way, the contextual probability is given by a call to `pc_cc/4` (with arguments `Class3`, `Class1` and `Class2`), rather than `pc_c/3`. However, in order to get the right results it is crucial to insert the goal `pc(Class2,_)` outside the innermost `findall`. The effect of this goal, in conjunction with the goal `pw_c(Word,Class3,LexP)` is that we find the best paths for every *pair of tags* `Class2`, `Class3` such that `Class3` is a possible

tag for `Word` and `Class2` is any tag. Without this or some equivalent goal, we would not get the best paths for every possible *state*, only for every *tag*. This might result in some candidate path being discarded — because it had lower probability for the last tag — even though it could later be extended into the best path for the entire input. In other words, the algorithm would no longer be guaranteed to find the best overall path.

While this implementation is perfectly adequate for the triclass tagging model, as usually described, it is actually not a correct implementation of the Viterbi algorithm for second-order HMMs. The reason is that output probabilities are not strictly speaking conditioned on states anymore but on tags, which really correspond to sets of states (where all the members of one set agree with respect to the second tag). It is of course an open question whether we actually want to make use of the possibility of conditioning words on pairs of tags because of the sparse data problem. But a correct implementation should at least allow this possibility. (If we do not want to use it we can simply define $P(w|s_i)$ to be the same for all the states s_i that agree with respect to their second tag.)

3.3.3 Second-Order Viterbi

The full second-order HMM requires lexical probabilities of a form that we have not encountered so far:

$$P(w_i|c_{i-1}, c_i)$$

In this model, every word is conditioned on one state, which in the second-order model corresponds to a pair of tags (in this case the tag of the word itself and the tag of the preceding word). We will not implement this lexical model until chapter 4, but we will assume for the remainder of this section, that they are defined by the lexical module in terms of the predicate `pw_cc/4` (in analogy with the predicate `pw_c/3` but with an extra argument for a second tag). The predicate `paths/3` can now be defined as follows:

```
paths([],MPPs,MPPs).

paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_cc(Word,Class2,Class3,LexP),
            findall(P1-[Class3,Class2,Class1|RestPath],
                    (member(P0-[Class2,Class1|RestPath],PPs0),
                    pc_cc(Class3,Class1,Class2,ConP),
                    P1 is P0*LexP*ConP),
                PPs1),
            most_probable(PPs1,PP)),
        PPs2),
    paths(Words,PPs2,PPs).
```

This program is very similar to the original biclass implementation, except for the fact that goals with `pw_cc/4` and `pc_cc/4` as main functors have been substituted for goals with `pw_c/3` and `pc_c/3`, respectively, and that the partially specified lists have one more element than before. And unlike the implementation in the preceding section, it allows us to exploit the full power of the second-order model, conditioning every word on a pair of tags. It should be

noted, however, that if we do not want to use this option, then the earlier implementation is still preferable, since it does not require us to define probabilities of the form $P(w_i|c_{i-1}, c_i)$.

3.4 Input/Output Tools

Minimally, the I/O module must be able to perform the following two tasks:

- Convert input text to lists of words (atoms) that can be passed to the search module via the predicate `most_probable_path/2`.
- Convert lists of words and lists of tags returned by the search module to whatever format is required as output from the tagger.

The details of these tasks will differ greatly from one system to the next depending on, among other things, the application in which the tagger is embedded and the input and output formats. Therefore, we will not spend much time on the development of input/output tools but mostly keep the discussion on a more general level.

3.4.1 Input Processing

There are three main types of input processing performed in a tagging system:

- Tokenization
- Segmentation
- Pre-processing

Tokenization Tokenization is the task of splitting up the text stream into words, or tokens, each of which should be assigned a tag. For standard European writing systems, this task is not too difficult, since word boundaries are usually marked by whitespace in the input. However, there are some complicating factors that nevertheless make it into a non-trivial problem:

- Punctuation marks are sometimes parts of tokens, sometimes tokens of their own. For example, a period (.) is a token when it functions as a full stop, but only part of a token in an abbreviation.
- Many non-alphabetic symbols have an unclear token status. For example, is *#1* one or two tokens? And what about dates like *08-07-2000*?
- Real texts often contain errors, which may give rise to erroneous tokenizations. For example, hyphens (which are part of tokens as in *part-of-speech*) and dashes (which are tokens of their own) are normally distinguished by the fact that the latter but not the former are surrounded by spaces (as in *Tagging — the ultimate challenge.*) However, in real texts spaces are often omitted or inserted, which means that we may end up with tokens like *Tagging-the* and *of-speech*.

- Finally, there are multi-word units, such as *in spite of*, which really should be given a single part-of-speech tag (in this case `pp` for preposition), even though they are written as three separate words. However, this is still beyond the scope of most taggers and therefore usually not considered as a tokenization problem.

In the following, we will bypass the problem of tokenization by working with pre-tokenized input (taken from existing corpora). Therefore, we will not discuss the implementation of tokenizers in Prolog. The interested reader is referred to [O'Keefe 1990] and [Covington 1994].

Segmentation Segmentation (or chunking) is the task of splitting up the tokenized text into strings of appropriate size. In theory, there is nothing that prevents us from tagging an entire text as a single string (given the linear complexity of the Viterbi algorithm). However, in practice, we are better off splitting the text up into smaller segments, for two reasons:

- The probabilities associated with different tag sequences for a given word string get progressively smaller as the length of the string grows. At some point, these probabilities get smaller than the smallest real number that we can represent in the computer, which means that they will be rounded off to zero and all tag sequences will be considered equally probable. This is sometimes referred to as the *underflow* problem. The most straighforward way of dealing with this problem is to make sure that the text is segmented into strings that are short enough for underflow not to occur.

- Certain kinds of pre-processing (and post-processing) are facilitated if the strings passed to the search module correspond to linguistic units such as sentences. We will discuss this briefly below and more fully in chapter 4.

A simple way of solving both these problems is to use punctuation marks, more precisely major delimiters such as *.*, *?* and *!*, as end markers for strings to be passed to the search module, which means that the strings will usually correspond to complete sentences. This strategy is not fool-proof, since you may encounter sentences that are so long that the underflow problem can still occur. Moreover, not all texts are clearly structured into sentences with punctuation marks.[6] Nevertheless, this is the strategy that will be assumed in the following.

Pre-processing In addition to tokenization and segmentation, which have to be applied to the input, there are various forms of pre-processing that may improve the quality of the tagging. For example, we may want to decapitalize the first word of every sentence to prevent it from being treated as an unknown word in case it doesn't occur capitalized in the training corpus. We may also want to run special algorithms to recognize special kinds of expressions such as dates, which may otherwise cause problems in tagging. We will return to some of these issues when we discuss optimization in the next chapter.

[6] A famous example is the last chapter of James Joyce's *Ulysses*, which runs to some sixty pages.

3.4.2 Output Processing

In our implementation, the search module returns pairs consisting of a word list and a tag list. In order to produce tagged text as output, the I/O module must therefore perform the inverse of segmentation and tokenization, i.e. it must piece the text together again, interleaving the part-of-speech annotation according to some pre-specified format. In addition, there may be a need for post-processing, such as capitalizing the first word of every sentence if it was decapitalized before passing the text to the search module.

3.4.3 Input/Output Tools

Having discussed the tasks of the I/O module at a fairly abstract level, we will now present a concrete Prolog implementation of the I/O module with enough functionality to allow us to build a working tagger. We will presuppose that the input has already been tokenized (but not segmented) and that each token occurs on a separate line in the text files that are input to the system. Output will be written in a similar format with each token starting a new line being followed by a tab and its part-of-speech tag.

The main predicate of the I/O module is `tag/2`, which is defined in the following way:

```
tag(InputFile,OutputFile) :-
     seeing(OldInputFile),
     telling(OldOutputFile),
     see(InputFile),
     tell(OutputFile),
     repeat,
          read_string(Words,EndOfFile),
          most_probable_path(Words,_P-Tags),
          write_tagged_string(Words,Tags),
          EndOfFile == yes,
     !,
     seen,
     told,
     see(OldInputFile),
     tell(OldOutputFile).
```

The predicate `tag/2` takes two arguments, an input file and an output file.[7] It begins by setting the input stream to be the input file (`see(InputFile)`) and the output stream to be the output file (`tell(OutputFile)`) and then starts the main loop (`repeat`). At each iteration of the loop, a string `Words` is read from the input file using the predicate `read_string/2` and passed to the search module via the predicate `most_probable_path/2`. After the search module returns a tag sequence `Tags`, the tagged string is written to the output file using the predicate `write_tagged_string/2`. This loop is repeated until the end of the input file is reached, after which the files are closed and the input/output streams restored to their previous values (`OldInputFile`, `OldOutputFile`).

[7] These are logical text files that may correspond to actual text files or to terminal input/output. The latter will be achieved by giving the argument `user` in most Prolog implementations.

The predicate `read_string/2` reads word by word, collecting the words in a list, until it encounters a major delimiter (*.*, *?* or *!*) or end-of-file. In the latter case, its second argument is instantiated to `yes`, which means that the exit condition of the loop is satisfied. The predicate `write_tagged_string/2` produces lines of text consisting of a word from the list `Words`, a tab, a tag from the list `Tags`, and a newline, calling itself recursively until the lists are exhausted. The implementation details of these two predicates can be found in appendix F.

3.5 Putting Things Together

In sections 3.1–3.4, we have presented tools for building each of the major components of a probabilistic part-of-speech tagger. It therefore seems appropriate at this point to say a few words about how they can be put together into a working system.

As mentioned several times, logic programs have the advantage of being fully incremental, in the sense that if we merge two logic programs (or simply compile them together) the result is a new logic program. Thus, all that needs to be done in order to build a complete system is to compile the required components together.[8] More precisely, we need the following:

- An I/O module that defines the predicate `tag/2`, which is used to run the tagger. Currently only one version of this module is available, which is contained in the file `io.pl` (cf. appendix F).

- A search module that defines the predicate `most_probable_path/2`, which is used to find the most probable tag sequence for a given word string according to the chosen probabilistic model. Three different implementations of this module exist (cf. appendix E):

File	**Description**
`viterbi.bi.pl`	Viterbi for biclass model
`viterbi.tri1.pl`	Viterbi for triclass model
`viterbi.tri2.pl`	Viterbi for second-order model

  Note, however, that the Viterbi algorithm for the full second-order model requires lexical probabilities of the form $P(w_i|c_{i-1}, c_i)$, which are not provided by the lexical modules defined in this chapter (cf. chapter 4).

- A lexical module that defines the predicate `pw_c/3`, which is used by the search module to access lexical probabilities. Four different implementations of this module exist (cf. appendix C):

File	**Description**
`lex.mle.pl`	Maximum likelihood estimation
`lex.uni.pl`	MLE with uniform model for unknown words
`lex.add.pl`	MLE with additive smoothing
`lex.gt.pl`	Good-Turing estimation

[8] This is true as long as we are content to run the tagger within the Prolog environment. Building stand-alone applications is of course slightly more complicated.

- A contextual module that defines the predicates `pc_c/3` and `pc_cc/4`, which are used by the search module to access contextual probabilities. Five different implementations of this module exist (cf. appendix D):

File	Description
`con.mle.pl`	Maximum likelihood estimation
`con.add.pl`	MLE with additive smoothing
`con.gt.pl`	Good-Turing estimation
`con.gt.pl`	Backoff smoothing
`con.gt.pl`	Linear interpolation

- A database module containing the following four frequency databases:

File	Description
`fwc.pl`	Word-tag frequencies
`fc.pl`	Tag frequencies
`fcc.pl`	Tag bigram frequencies
`fccc.pl`	Tag trigram frequencies

  We must also define the set of open classes in a special database:

File	Description
`open.pl`	Open word classes (`open(`c`)`)

  If Good-Turing estimation is used, then the following databases defining reestimated frequencies must also be supplied:

File	Description
`gtfwc.pl`	Reestimated word-tag frequencies
`gtfcc.pl`	Reestimated tag bigram frequencies
`gtfccc.pl`	Reestimated tag trigram frequencies

Finally, there is a small number of facts that must be asserted into the internal database:

- The number of tokens in the training corpus is given by `tokens(`n_{token}`)`.
- The number of types in the training corpus is given by `types(`n_{type}`)`.
- The number of tags in the training corpus is given by `classes(`n_{class}`)`.
- The frequencies for tag n-grams involving the dummy tag `start` are given by facts `fc(start,`f_0`)`, `fcc(start,`c_i`,`f_{c_i}`)`, `fccc(start,start,`c_i`,`f_{c_i}`)` and `fccc(start,`c_i`,`c_j`,`$f_{(c_i,c_j)}$`)` (cf. section 3.2).
- If additive smoothing is used for the lexical model, the additive constant is given by `lex_add(`k_{lex}`)`.
- If additive smoothing is used for the contextual model, the additive constant is given by `con_add(`k_{con}`)`.
- If backoff smoothing is used for the contextual model, the frequency threshold and discounting factor are given by the facts `threshold(`t`)` and `discount(`d`)`, respectively.

- If linear interpolation is used for the contextual model, the interpolation weights are given by the facts `bi1(`λ_{bi1}`)`, `bi2(`λ_{bi2}`)`, `tri1(`λ_{tri1}`)`, `tri2(`λ_{tri2}`)` and `tri3(`λ_{tri3}`)`.

In order to simplify the compilation of different components and the assertion of facts into the internal database, we provide a program `main.pl` which, when compiled, automatically compiles the required databases and the I/O module. It also computes and asserts the number of tokens, types and classes, and defines frequencies involving the dummy tag `start` according to the simple scheme outlined in section 3.2.[9] This produces a complete system except for the lexical and contextual moduls. However, the program `main.pl` also provides a predicate `setup/0`, which can be used to interactively specify the following parameters:

- Order of HMM and Viterbi implementation (`bi`, `tri1`, `tri2`)
- Lexical model (`mle`, `uni`, `add`, `gt`)
- Contextual model (`mle`, `add`, `gt`, `back`, `int`)

Given the choice of lexical and contextual model, the user is prompted for values of relevant parameters (additive constants, frequency thresholds, etc.), which are asserted into the internal database. Thus, by compiling `main.pl` and running `setup/0`, the user can build a complete tagging system. The implementation details of `main.pl` can be found in appendix G.[10]

[9] The computation of dummy tag frequencies does not work in SICStus Prolog. Instead, the corresponding rules have to be inserted into the files `fc.pl`, `fcc.pl` and `fccc.pl`, respectively; see `main.pl` for details.

[10] This program can be run in SWI-Prolog and (with minimal modifications described in the source file) SICStus Prolog. For other Prolog implementations, minor modifications may be necessary to make sure that, for example, built-in predicates for list processing are available.

Chapter 4

Sharpening the Tools

In the previous chapter, we have presented all the tools that are needed to build a probabilistic part-of-speech tagger in a logic programming framework. By combining a suitable collection of components, as explained in section 3.5, we can build a fully functional tagging system that can be used to analyze any kind of text. However, such a system will be less than optimal in two important respects:

- **Accuracy:** The tagger will make unnecessary errors because no special techniques are used to deal with, for example, unknown words. Therefore, the tagging accuracy will be suboptimal.
- **Efficiency:** The tagger will be slowed down by the fact that probabilities are computed from raw frequencies at runtime. Moreover, in tagging any sizeable amount of text, the same probabilities will be computed more than once. Therefore, the tagging speed will be suboptimal.

In this chapter, we will discuss different ways in which both accuracy and efficiency can be improved. We will begin, in section 4.1, by defining a baseline tagger, i.e., a tagger built only from tools presented in the preceding chapter, which will be used as a standard of comparison when evaluating different optimization methods. In section 4.2, we will then discuss different ways of improving tagging accuracy, while section 4.3 will be devoted to techniques for enhancing the speed of the tagger.

4.1 Baseline

The baseline tagger uses the standard triclass tagging model without conditioning words on pairs of tags. The contextual model is smoothed using additive smoothing with $k = 0.5$, while the lexical model using Good-Turing estimation. This particular configuration has been shown to give comparatively good results, keeping in mind that no optimization at all has been performed yet ([Nivre 2000]).

Ten blocks of 1115 words each[1] were randomly drawn from the Stockholm-Umeå Corpus (SUC), a tagged corpus of written contemporary Swedish, in

[1] In fact, the last sample only contains 1113 words.

order to be used as test data. The remaining 1155774 words, constituting 99% of the entire corpus were then used as training data to derive the frequency databases `fc.pl`, `fcc.pl`, `fccc.pl` and `fwc.pl`. The number of facts in each of these databases is given in Table 4.1. It should be noted that no normalization of word forms was performed, which means, for example, that word forms differing only with respect to capitalization were considered as different word types. We will return to this problem in section 4.2.2 below.

Table 4.1: Number of facts in frequency databases derived from SUC

Database	`Facts`
`fc.pl`	23
`fcc.pl`	458
`fccc.pl`	5299
`fwc.pl`	110421

The tagset used was the basic SUC tagset, without features, consisting of the 23 tags and described in appendix A. Of these 23 tags, nine were treated as open classes during tagging and allowed as tags for unknown words. The tags corresponding to open classes are listed in table 4.2.

Table 4.2: Open classes (9 tags)

Tag	**Part-of-speech**	**Example**
`ab`	Adverb	*här, sakta*
`in`	Interjection	*hej, usch*
`jj`	Adjective	*gul, varm*
`nn`	Noun	*bil, kvinna*
`pc`	Participle	*springande*
`pm`	Proper name	*Lisa, Växjö*
`rg`	Cardinal numeral	*tre, 2000*
`uo`	Foreign word	*the, yes*
`vb`	Verb	*går, leka*

When scoring the accuracy of the tagger, the tags provided by the SUC corpus were always considered correct, even though the SUC corpus is known to contain a small percentage of tagging errors. The baseline tagger, when trained on 99% of the SUC corpus and tested on the remaining 1% achieved an accuracy rate of 94.64% (598 errors in 11148 words). The time required to tag the test corpus of 11148 was 222.58 seconds (CPU time), which corresponds to a tagging rate of about 50 words per second.

4.2 Improving Accuracy

Many of the errors performed by part-of-speech taggers in general and probabilistic taggers in particular are caused by the presence of unknown words in

the input, i.e., word forms that have not occurred in the training data.[2] It is important to remember that an inadequate treatment of unknown words may induce errors not only in the tagging of the unknown words themselves, but also in the tagging of neighboring words because of inaccurate contextual information. Since the Viterbi algorithm maximizes probability over the entire string, this may affect long chains of words.

Most of our attempts at improving the accuracy rate of our baseline tagger will therefore be targeted at unknown words. More precisely, we will discuss the following problem areas:

- **Numerals:** These words form an open-ended but syntactically definable class, which means that previously unseen members can nevertheless be recognized by a parser (cf. section 4.2.1).
- **Capitalization:** Proper names are usually capitalized in Swedish and other languages, but so are words occurring at the start of a sentence or in a headline. Moreover, capitalization is often inconsistent in input texts. This means, among other things, that a word may be unknown simply because it occurs at the start of a sentence (if it occurs only without capitalization in the training corpus) and that capitalization is not a reliable indication that an unknown word is a proper name (cf. section 4.2.2).
- **Word endings:** Quite often, unknown words have recognizable subparts which may indicate what part-of-speech they belong to. For example, if the word *ekskrivbord* (oak desk) does not occur in our Swedish training corpus but the word *skrivbord* (desk) does, then we can recognize the latter as a proper suffix of the former and conjecture that they have the same part-of-speech, which in this case would be correct (cf. section 4.2.3).

However, not all tagging errors are related to unknown words. Another serious problem is constituted by high-frequency words with multiple tags that are difficult to distinguish, such as the Swedish word *det* (the/it/that) which may function both as determiner (*det röda huset* 'the red house') and as pronoun (*det är rött* 'it is red'). In fact, in the study of [Nivre and Grönqvist forthcoming], as few as eight word forms accounted for more than 30% of all tagging errors. Since high-frequency words will be amply represented in the training corpus, they are fairly insensitive to the sparse data problem, which means that it may be possible to improve the probabilistic model by conditioning their lexical probabilities on pairs of tags (as opposed to single tags), in accordance with the full second-order Markov model discussed in section 3.3.3. This possibility, which to our knowledge has not been discussed in the literature, will be explored in section 4.2.4. Finally, in section 4.2.5, we will try to combine all the optimization techniques discussed in sections 4.2.1–4.2.4 to see how much they improve the overall tagging accuracy.

[2] In addition to unknown words, we may also encounter known words with unknown tags, i.e., word forms that do occur in the training data but not with the appropriate tag. For most taggers it is simply impossible to tag these words correctly, since known words are usually only allowed to have known tags. Fortunately, these words are fairly infrequent if the training corpus is sufficiently large. In the study of [Carlberger and Kann 1999], they account for 3.9% of all errors made by the tagger. In [Nivre 2000], the corresponding figure was close to 5% for some tagging configurations.

4.2.1 Numerals

Cardinal numerals, i.e., words denoting numbers such as *ett* (one), *fem* (five), *fjorton* (fourteen), *tjugoett* (twenty-one), etc., form a potentially infinite class (at least in a language such as Swedish where they are written without spaces). Nevertheless, they are always built up from the same basic elements (morphemes such as *två* [two], *tio* [ten], *hundra* [hundred], *tusen* [thousand], etc.) and have a reasonably well-defined syntactic structure. Therefore, even though no training corpus will ever be complete with respect to numerals, we can construct a parser that recognizes numerals even if they do not occur in the training corpus.

Below we define a predicate `numeral/1`, which is true of a word form if it is a numeral. The definition consists of two clauses, the first of which covers the case where the numeral is written with digits (*1, 5, 14, 21*, etc.):

```
numeral(W) :-
     atom_chars(W,[C|Cs]),
     47 < C, C < 58,
     !,
     no_letters(Cs).
```

This clause says that a word `W` is a numeral if its first character `C` is a digit (i.e., has an ASCII code between 48 and 57 inclusive), and if the rest of the word does not contain any letters. This means that also numerical expressions such as *3+5* will count as numerals. (The built-in predicate `atom_chars/2` maps an atom to the list of its characters, represented as ASCII codes. The predicate `no_letters/1` runs through a list of characters and succeeds if none of the characters is a letter.)

The second clause defining the predicate `numeral/1` deals with the case where the numeral is written with letters (*ett, fem, fjorton, tjugoett*, etc.):

```
numeral(W) :-
     atom_chars(W,Cs),
     nums(Cs).

nums([]).
nums(S) :-
     split(S,S1,S2),
     num(S2),
     nums(S1).
```

The first clause says that a word is a numeral if the list of its characters satisfies the `nums/1` predicate. The latter is true of a list of characters if a suffix of the list forms a numeral morpheme (`num/1`) and the remaining prefix itself satisfies the `nums/1` predicate. Thus, `nums/1` will recursively decompose a list of characters into numeral morphemes (starting at the end) until the list is empty.[3] In order for this program to work as a numeral parser, we must also supply a database of numeral morphemes, the beginning of which may look as follows:

```
num("en").
num("ett").
```

[3] This is not the most efficient way to solve this problem but quite sufficient for our current purposes.

```
num("två").
num("tre").
num("fyra").
num("fem").
...
```

Note the use of double quotes in order for the arguments to be interpreted as strings (lists of characters) and not as atoms (as required by the predicate `nums/1`).

Given a numeral parser like the one sketched above, we can modify the definition of lexical probabilities as follows (cf. section 3.1.4:

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     \+ open(C),
     fc(C,F2),
     P is F1/F2.

pw_c(W,C,P) :-
     fwc(W,C,F1),
     open(C),
     gtfwc(C,F1,F2),
     fc(C,F3),
     P is F2/F3.

pw_c(W,rg,P) :-
     \+ fwc(W,_,_),
     numeral(W),
     unseen_wc(rg,F1),
     fc(rg,F2),
     P is F1/F2.

pw_c(W,C,P) :-
     \+ fwc(W,_,_),
     \+ numeral(W),
     open(C),
     unseen_wc(C,F1),
     fc(C,F2),
     P is F1/F2.
```

The only new clause is the third one, which says that if a word is unknown (does not occur in the frequency database `fwc.pl`) and if it can be parsed as a numeral, then it is assigned the probability reserved for unknown numerals by Good-Turing estimation. In addition, the fourth and final clause has been modified so that it only applies to unknown words that can not be parsed as numerals. The combined effect is that an unknown word which can be parsed as a numeral will always be tagged as a numeral (since its lexical probability is undefined for all other tags).

Adding the numeral parser and the revised definition of lexical probability to our baseline tagger results in a small improvement of tagging accuracy, from 94.64% to 94.73% (587 errors in 11148 words, as opposed to 598 for the baseline

tagger). The difference is small but nevertheless statistically significant beyond the .01 level (McNemar's test[4]), which means that it is consistent. Thus, the main reason why the improvement is small is that unknown numerals are infrequent. The time required to tag the test corpus with the extended tagger was 220.85 seconds, which is about the same as for the baseline tagger.[5] In other words, the addition of the numeral parser does not slow down the tagger noticeably.

4.2.2 Capitalization

Basically, there are two main considerations concerning capitalized words in tagging:

- The first word in a sentence is usually capitalized regardless of its part-of-speech. Such a word may therefore appear to be unknown, even though it occurs without capitalization in the training corpus.
- Capitalized words in other positions are likely to be proper names, regardless of whether they are unknown or not.

The first of these problems is handled straightforwardly by a simple modification to the I/O module:

```
tag(InputFile,OutputFile) :-
     seeing(OldInputFile),
     telling(OldOutputFile),
     see(InputFile),
     tell(OutputFile),
     abolish(runtime,1),
     assert(runtime(0)),
     repeat,
             read_string(Words,EndOfFile),
             decap(Words,DWords),
             most_probable_path(DWords,_P-Path),
             write_tagged_string(Words,Path),
             EndOfFile == yes,
     !,
     seen,
     told,
     see(OldInputFile),
     tell(OldOutputFile).
```

Before passing the input string `Words` to the search module via the predicate `most_probable_path/2`, we filter it through the predicate `decap/2`, defined in the following way:

```
decap([Word|Words],[DWord|Words]) :-
     \+ fwc(Word,_,_),
```

[4]The use of McNemar's test instead of the more common paired t-test is motivated by considerations in [Dietterich 1998]. For a description of McNemar's test, see, e.g., [Everitt 1977].

[5]In fact, it is slightly less than for the baseline tagger, but this difference is probably not significant.

```
        !,
        atom_chars(Word,Cs),
        decapitalize(Cs,Ds),
        atom_chars(DWord,Ds).

decap(W,W).
```

If the first word `Word` is unknown, then it is decapitalized; otherwise it is left as it is. (The predicate `decapitalize/2` runs through a list of characters and changes any uppercase character to the corresponding lowercase character.) The reasoning behind this treatment is the following:

1. If the word is known (and capitalized), it is either a known proper name or a known word that occurs in sentence-initial position in the training corpus (or both). In this case, we are better off leaving the word as it is, since decapitalization may remove some of the available tags.

2. If the word is unknown (and capitalized), it is most likely that the word is not a proper name and that it may therefore occur without capitalization in the training corpus. Therefore, we decapitalize it before consulting the lexical module.

Note that the predicate `write_tagged_string/2` still takes the original string as its first argument. Hence, there is no need for post-processing in order to restore the capitalization of the first word.

The second problem involving capitalization requires a modification of the lexical module. As a first attempt, we might propose the following:

```
pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,pm,P) :-
        \+ fwc(W,_,_),
        cap(W),
        !,
        unseen_wc(pm,F1),
        fc(pm,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
```

```
        unseen_wc(C,F1),
        fc(C,F2),
        P is F1/F2.

cap(W) :-
        atom_chars(W,[C1,C2|_Cs]),
        uppercase(C1),
        \+ uppercase(C2).
```

This treatment would be parallel to the treatment of numerals in the preceding section in that it would force an unknown word to be tagged as a proper name as soon as it was capitalized. However, this solution is indequate for two reasons. First, it fails to take into account the fact that words may be capitalized even if they are not proper names (e.g., when they occur in headlines). Such words would then be erroneously classified as proper names. Secondly, since the class of proper names is an open class, the fourth and final clause may allow a word to have a high lexical probability for the tag `pm` (proper name) even if it is not capitalized, which in most cases is not a good idea.

We will therefore follow [Carlberger and Kann 1999] in allowing capitalized unknown words to have other tags besides `pm`, and also allow non-capitalized words to be tagged as proper names, but in both cases with a penalty, which consists in multiplying the estimated probability with a constant c ($0 < c < 1$). [Carlberger and Kann 1999] found that the constants 0.02 (for capitalized words with tags other than `pm`) and 0.05 (for non-capitalized proper names) seemed to give the best performance. These are the constants that are used in our program as well:

```
pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
        unseen_wc(C,F1),
        fc(C,F2),
        P1 is F1/F2,
        ( cap(W) -> ( \+C==pm -> P is 0.02*P1 ; P is P1 ) ;
          ( C==pm -> P is 0.05*P1 ; P is P1) ).

cap(W) :-
        atom_chars(W,[C1,C2|_Cs]),
```

```
        uppercase(C1),
        \+ uppercase(C2).
```

The third clause (which replaces the third and fourth clause in the previous formulation) uses Prolog's if-then-else-construct to specify that the estimated probability should be multiplied by 0.02 if the word is capitalized and the tag is different from `pm`, multiplied by 0.05 if the word is not capitalized and the tag is `pm`, and left intact otherwise.

Using the new treatment of capitalized word (both the new I/O module and the new lexical module) increases the accuracy rate of the baseline tagger from 94.64% to 95.21% (534 errors in 11148 words, as opposed to 598), which is a substantial improvement and a 10% reduction of the error rate. Again, the difference is statistically significant beyond the .01 level (McNemar's test). It took 213.72 seconds of CPU time to tag the test corpus, which means that tagging speed is comparable to that of the baseline tagger (222.58 seconds).

4.2.3 Word Endings

In section 4.2.1, we saw how numerals could be recognized by being segmented into recognizable subparts. The same general strategy can be applied to unknown words in general, since most unknown words have parts that are known words or parts of known words. In a language like Swedish, it is most informative to consider the endings of words. First of all, parts-of-speech such as nouns, verbs, adjectives and adverbs have typical suffixes (inflectional as well as derivational). Moreover, many unknown words are compounds, consisting of two or more words, and the word class of the compound is typically determined by the word class of the last constituent word.

In order to test this methodology, we first built a frequency database for word suffixes from our training corpus as follows:

fsc.pl The frequency of the suffix s occurring with the tag c, i.e., $f(s,c)$, expressed in clauses of the form `fsc(s,c,n)`, where $n = f(s,c)$.

Since the training corpus contained no morphological analysis, all possible suffixes were considered. Thus, if the word *dator* (computer) occurred 35 times with tag `nn` (noun), then the suffixes *ator*, *tor*, *or*, and *r*, all had their frequency count for the tag `nn` incremented by 35. However, to prevent the database from getting too large, suffixes with frequency 1 were omitted, which reduced the size of the database from 341639 clauses to 139258 clauses.[6]

Given such a frequency database, it is possible to define lexical probabilities for unknown words in many different ways, some more sophisticated than others (see, e.g., [Samuelsson 1994, Brants and Samuelsson 1995]). Here we will consider one of the simplest methods, which basically consists in finding the longest possible suffix that occurs in the suffix frequency database and using the MLE lexical probabilities for this suffix as the lexical probabilities of the unknown word. In addition, we will check whether the longest suffix also occurs as a separate word, in which case the lexical probabilities for this word will be permitted as well (in order to recognize compound words). This strategy is implemented by the following logic program:

[6] This can be compared to the word-tag frequency database, which consists of 110421 clauses.

```
pw_c(W,C,P) :-
     fwc(W,C,F1),
     \+ open(C),
     fc(C,F2),
     P is F1/F2.

pw_c(W,C,P) :-
     fwc(W,C,F1),
     open(C),
     gtfwc(C,F1,F2),
     fc(C,F3),
     P is F2/F3.

pw_c(W,C,P) :-
     \+ fwc(W,_,_),
     unknown_prob(W,C,P).

unknown_prob(W,C,P) :-
     ending(E,W),
     (fwc(E,C,F1) ; fsc(E,C,F1)),
     open(C),
     fc(C,F2),
     P is F1/F2.

unknown_prob(W,C,P) :-
     \+ ending(_,W),
     open(C),
     unseen_wc(C,F1),
     fc(C,F2),
     P is F1/F2.

ending(E,W) :-
     atom_chars(W,Ws),
     suffix(Es,Ws),
     atom_chars(E,Es),
     fsc(E,_,_),
     !.
```

The first two clauses for the predicate `pw_c/3` are the same as for ordinary Good-Turing estimation. The third clause simply says that for unknown words lexical probabilities are given by the auxiliary predicate `unknown_prob/3`. The latter predicate has two clauses, the first of which says that the lexical probability for (unknown) word `W` with tag `C` is `P` if `E` is the longest known ending of `W`, if `E` has frequency `F1` with tag `C`, either as a word (`fwc(E,C,F1)`) or as a suffix (`fsc(E,C,F1)`), if `C` is an open word class, and if `P` is the result of dividing `F1` with the tag frequency `F2` of `C`. The second clause for `unknown_prob/3` is a catch-all clause for words where no known ending can be found (which almost never happens in practice if one-character suffixes are allowed), and defines such words to have the probabilities given by ordinary Good-Turing estimation for unknown words. Finally, the predicate `ending/2` is defined in such a way that it

succeeds at most once, always instantiating the variable `E` to the longest known suffix of the given word `W`.

Adding this treatment of unknown words to the baseline tagger increases the accuracy rate from 94.64% to 95.05% (552 errors in 11148 words, as opposed to 598), a difference which is significant beyond the .01 level (McNemar's test). Moreover, the time to tag the test corpus went down from 222.58 seconds to 93.50 seconds of CPU time, which is a reduction of more than 50%. Thus, the use of word endings in the analysis of unknown words improves not only accuracy but also efficiency. The reason is that unknown words have fewer possible tags, since only the tags of the longest suffix are considered, whereas with the old strategy every unknown word can have every open-class tag.

4.2.4 Second-Order Lexical Models

As noted several times (cf. sections 3.1 and 3.3.3), the standard triclass tagging model deviates from the full second-order Markov model in conditioning lexical probabilities on single tags instead of the tag pairs represented by states of the Markov model. The main reason for this deviation is that the size of the probabilistic model can become forbiddingly large, which may have a negative influence both on accuracy, because of the sparse data problem, and on efficiency, because of the larger search space. Given a lexicon of m words and a tagset of n tags, the full second-order lexical model will have mn^2 parameters, as compared to the mn parameters of the ordinary lexical model. For example, given our training corpus containing 110421 distinct word types and our limited tagset of 23 tags, the number of parameters in the second-order model is 58412709, to be compared with 2539683 for the ordinary first-order model. For larger tagsets, the difference will be greater still.

Nevertheless, the second-order model is more fine-grained in that it takes a larger amount of context into account for lexical probabilities, and could therefore improve accuracy, at least for high-frequency words where probability estimates are more reliable. In order to test this hypothesis, which to our knowledge has not been addressed in the literature, we first built a second-order frequency database from our training corpus as follows:

fwcc.pl The frequency of word w_i with tag c_i, preceded by a word with tag c_{i-1}, i.e., $f(w_i, c_{i-1}, c_i)$, expressed in clauses of the form `fwc(`w`,`c_{i-1}`,`c_i`,`n`)`, where $n = f(w_i, c_{i-1}, c_i)$.

Given this frequency database, the definition of second-order lexical probabilities $P(w_i|c_{i-1}, c_i)$ can be implemented by the following logic program:

```
pw_cc(W,C1,C2,P) :-
     fwcc(W,C1,C2,F1),
     hfreq(F),
     F1 > F,
     fcc(C1,C2,F2),
     P is F1/F2.

pw_cc(W,C1,C2,P) :-
     pw_c(W,C2,P),
     fc(C1,_),
```

```
      \+ (fwcc(W,C1,C2,F1), hfreq(F), F1 > F).

pw_c(W,C,P) :-
      fwc(W,C,F1),
      \+ open(C),
      fc(C,F2),
      P is F1/F2.

pw_c(W,C,P) :-
      fwc(W,C,F1),
      open(C),
      gtfwc(C,F1,F2),
      fc(C,F3),
      P is F2/F3.

pw_c(W,C,P) :-
      \+ fwc(W,_,_),
      open(C),
      unseen_wc(C,F1),
      fc(C,F2),
      P is F1/F2.
```

The first clause says that the lexical probability of word `W` given tags `C1` and `C2` is a pure maximum likelihood estimate if their joint frequency `F1` is higher than a pre-defined frequency threshold given by the predicate `hfreq/1`. The second clause takes care of the case where the joint frequency is below the threshold, in which case the ordinary lexical probability of `W` given `C1` is used as an estimate for the second-order probability.[7] Formally, this can be expressed as follows (where t is the frequency threshold):

$$\hat{P}(w_i|c_{i-1},c_1) = \begin{cases} \frac{f(w_i,c_{i-1},c_i)}{f(c_{i-1},c_i)} & \text{if} f(w_i,c_{i-1},c_i) > t \\ \hat{P}(w_i|c_i) & \text{otherwise} \end{cases}$$

The remaining clauses in the above program are the same as for the ordinary lexical model using Good-Turing estimation.

In order to use the second-order lexical model, we also need to use the second-order version of the Viterbi algorithm, as defined in section 3.3.3. Substituting these programs for the corresponding components in the baseline tagger, and setting the frequency threshold at 10, results in a small but significant ($p < 0.01$) improvement of the accuracy rate from 94.64% to 94.94%. The time required to tag the test corpus was 209.30, which compares favorably with the result for the baseline tagger (220.85 seconds).

4.2.5 Putting It All Together

In sections 4.2.1–4.2.4, we have considered four different ways of improving the accuracy rate of a probabilistic part-of-speech tagger, all of which have resulted

[7] Note that clause order in the body of the second rule is crucially important here. Since negation as failure does not bind variables, the negative condition must be placed last, otherwise the clause would only apply to words whose frequency does not exceed the threshold for *any* tag pair.

in significant improvements when tested on the available data. However, it remains to see whether it is possible to combine them all in order to improve accuracy even further. At first, this may seem self-evident, but we will see that the effects of different techniques actually can cancel each other out. In any case, we should not expect the combined effect to be a simple sum of the effects observed for the individual methods.

In order to combine all the techniques discussed in previous sections, we first need to make the following modifications to our baseline tagger:

- Replace the old I/O module by the new I/O module handling capitalization (cf. section 4.2.2).
- Replace the triclass search module by the second-order search module (cf. sections 3.3.3 and 4.2.4).
- Include the parser for numerals (cf. section 4.2.1).
- Include the suffix frequency database (cf. section 4.2.1).
- Include the second-order lexical database (cf. section 4.2.4).

Finally, we need to define a new lexical module in order to combine the different treatments of unknown words in a reasonable way:

```
pw_cc(W,C1,C2,P) :-
     fwcc(W,C1,C2,F1),
     F1 > 10,
     fcc(C1,C2,F2),
     P is F1/F2.

pw_cc(W,C1,C2,P) :-
     pw_c(W,C2,P),
     fc(C1,_),
     \+ (fwcc(W,C1,C2,F1), hfreq(F), F1 > F).

pw_c(W,C,P) :-
     fwc(W,C,F1),
     \+ open(C),
     fc(C,F2),
     P is F1/F2.

pw_c(W,C,P) :-
     fwc(W,C,F1),
     open(C),
     gtfwc(C,F1,F2),
     fc(C,F3),
     P is F2/F3.

pw_c(W,rg,P) :-
     \+ fwc(W,_,_),
     numeral(W),
     unseen_wc(rg,F1),
```

```
     fc(rg,F2),
     P is F1/F2.

pw_c(W,C,P) :-
     \+ fwc(W,_,_),
     \+ numeral(W),
     unknown_prob(W,C,P1),
     ( cap(W) -> ( \+C==pm -> P is 0.02*P1 ; P is P1 ) ;
       ( C==pm -> P is 0.05*P1 ; P is P1) ).

unknown_prob(W,pm,P) :-
     cap(W),
     unseen_wc(pm,F1),
     fc(pm,F2),
     P is F1/F2.

unknown_prob(W,C,P) :-
     ending(E,W),
     (fwc(E,C,F1) ; fsc(E,C,F1)),
     open(C),
     fc(C,F2),
     P is F1/F2.

unknown_prob(W,C,P) :-
     \+ ending(_,W),
     open(C),
     unseen_wc(C,F1),
     fc(C,F2),
     P is F1/F2.

cap(W) :-
     atom_chars(W,[C1,C2|_Cs]),
     uppercase(C1),
     \+ uppercase(C2).

ending(E,W) :-
     atom_chars(W,Ws),
     suffix(Es,Ws),
     atom_chars(E,Es),
     fsc(E,_,_),
     !.
```

The first two clauses implement the second-order lexical model (cf. section 4.2.4), using the first-order model when frequencies are below the pre-defined threshold. The next two clauses define first-order lexical probabilities for known words belonging to closed and open classes, respectively. The fifth clause takes care of unknown words that can be parsed as numerals (which are only allowed to have the tag `rg` [cardinal numeral]), and the sixth clause handles other unknown words taking capitalization into account. The auxiliary predicate `unknown_prob/3` has three clauses, one for capitalized words which are always allowed to have

the tag `pm` (proper name), one for words with known endings, and one for the remaining unknown words which can be assigned to any open class.

Using this combination of techniques improves the accuracy rate from 94.64% to 95.52%, which corresponds to an error rate reduction of about 16%. This difference, which is statistically significant beyond the .01 level (McNemar's test), is greater than for any of the single methods considered earlier but is far from the sum of the differences obtained for the individual methods (which would have meant an accuracy rate of 96.01%). The time required to tag the test corpus was 121.07 seconds, which is considerably faster than the baseline tagger (220.85 seconds) but slower than the tagger tested in section 4.2.3 (93.50 seconds).

An accuracy rate of 95.52% is slightly below the best results published for this kind of data. [Carlberger and Kann 1999] report an accuracy rate of 96.4% for data taken from the same corpus (SUC) with similar evaluation methods. On the other hand, we have only explored a limited set of optimization techniques in this chapter.

4.3 Improving Efficiency

So far in this chapter we have been concerned mainly with the improvement of tagging accuracy, which is arguably the most important aspect of optimization. At the same time, however, we have seen that some of the measures taken in order to improve accuracy can also be beneficial from the point of view of efficiency. In particular, we have seen that the use of suffix information to determine lexical probabilities for unknown words reduces running time by reducing the number of possible tags for unknown words, thereby also reducing the number of path extensions that have to be considered when an unknown word is encountered in the input string.

In this section, we will discuss two other ways to improve the speed of our tagging system:

- Improving the search algorithm for the second-order Markov model.
- Precompiling the contextual and lexical modules.

4.3.1 Improving the Search Algorithm

Let us return to the implementation of the Viterbi algorithm for second-order Markov models, discussed in section 3.3.3. The heart of this program is the definition of the predicate `paths/3`, and in particular the recursive clause involving two nested calls to the all-solutions predicate `findall/3`. We repeat the definition below for convenience:

```
paths([],MPPs,MPPs).

paths([Word|Words],PPs0,PPs) :-
    findall(PP,
         (pw_cc(Word,Class2,Class3,LexP),
             findall(P1-[Class3,Class2,Class1|RestPath],
                  (member(P0-[Class2,Class1|RestPath],PPs0),
```

```
                    pc_cc(Class3,Class1,Class2,ConP),
                    P1 is P0*LexP*ConP),
                   PPs1),
             most_probable(PPs1,PP)),
            PPs2),
    paths(Words,PPs2,PPs).
```

The first call to `findall/3` finds all states in which the first word of (the rest of) the input string can be read with non-zero probability, each state corresponding to a different instantiation of the pair (`Class2`,`Class3`). The second call to `findall/3` then finds all the candidate paths constructed for the preceding words in the input string and selects, for each state found in the first call, the path producing the maximum probability. These best paths are then passed on as candidate paths in the recursive call to `paths/3`.

The problem with this algorithm, when applied to the second-order Markov model, is that the first call to `findall/3` will find many states that are possible from the point of view of the current word, but which are nevertheless impossible given the candidate paths constructed for the preceding words in the input string. Since a path consisting of two states represent the tags of three words, the two states must always agree on the 'middle tag'. Formally, if $s_i = (c_{i1}, c_{i2})$ and $s_j = (c_{j1}, c_{j2})$ are states of the model, then (s_1, s_2) is a possible path only if $c_{i2} = c_{j1}$. But for every possible instantiation of `Class3` (i.e., for every possible tag of `Word`), the variable `Class2` will be instantiated to every possible tag, regardless of whether this tag occurs in any of the candidate paths in the list `PPs0`. Thus, if the preceding word has k possible tags, if the current word has m possible tags, and if there are n different tags in the tagset altogether, the algorithm will consider mn different states instead of the mk states that are really possible. For a tagset of the size used in the experiments here, the average value of k is about 2, while n is 23, which means that the number of path extensions considered by the above algorithm is on average one order of magnitude greater than the number of possible path extensions.

In order to remedy this problem, we need to introduce some kind of 'look-ahead' mechanism, so that the variable `Class2` is only instantiated to tags that are possible in the context of the preceding word. A straightforward way of doing this is to add the list of possible tags for the preceding word as an extra argument to the `paths` predicate as follows:

```
paths([],MPPs,_,MPPs).

paths([Word|Words],PPs0,Classes0,PPs) :-
    findall(PP,
            (member(Class2,Classes0),
             pw_cc(Word,Class2,Class3,LexP),
             findall(P1-[Class3,Class2,Class1|RestPath],
                  (member(P0-[Class2,Class1|RestPath],PPs0),
                   pc_cc(Class3,Class1,Class2,ConP),
                   P1 is P0*LexP*ConP),
                  PPs1),
             most_probable(PPs1,PP)),
            PPs2),
    setof(Class,P^Rest^member(P-[Class|Rest],PPs2),Classes2),
```

```
    paths(Words,PPs2,Classes2,PPs).
```

When the predicate `paths/4` is called, the variable `Classes0` will be instantiated to a list containing the possible tags of the preceding word, and before the call to `pw_cc/4`, the variable `Class2` will therefore be instantiated to one of these tags (via the call to `member/2`). Before the recursive call to `paths/4`, we construct the list of tags for the current word with a call to `setof/3`.

The modification of `paths/3` to `paths/4` also requires a modification of the top-level predicate `most_probable_path/2`:

```
most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start,start]],[start],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start,start|Path]).
```

The third argument in the call to `paths/4` is the list `[start]`, which is the list of all possible tags for the pseudo-word occurring before the first word in the input string.

Besides reducing the number of candidate paths that need to be considered for each new word in the input string, this modification of the algorithm has the advantage that the variable `Class2`, corresponding to the tag of the preceding word, will always be instantiated before the call to `pw_cc/4`, which means that the implementation of `pw_cc/4` can be simplified.

4.3.2 Precompiling Contextual and Lexical Modules

In the basic toolbox presented in chapter 3, the lexical and contextual modules are implemented as small logic programs defining probabilities in terms of raw corpus frequencies (provided by frequency databases). This is very convenient when the toolbox is used for pedagogical purposes, since it is easy to experiment with different implementations of these modules without changing the large databases. From a computational point of view, however, this means that many probabilities will be computed over and over again. Take the word *och* (and), for example, which occurs 302 times in the test corpus of 11148 words. Every time this word is encountered in the input, its lexical probabilities will be computed from scratch. Once we have decided to use a particular configuration of modules, it would therefore be much better if we could compute all probabilities beforehand, so that access to the lexical and contextual modules at run-time can be reduced, as far as possible, to pure database lookup.

Fortunately, this can be achieved very easily within the logic programming framework, using meta-programming techniques, since the probability databases that we want to use at run-time follow logically from the conjunction of our probability definitions and our frequency databases. For example, suppose we have the following facts in our frequency databases:

```
fwc(att,ie,12339).
fwc(att,sn,11289).

fc(ie,12637).
fc(sn,17012).
```

And suppose we have the following (simplified) definition of lexical probability:

```
pw_cc(W,C,P) :-
     fwc(W,C,F1),
     fc(C,F2),
     P is F1/F2.
```

Then the following facts are logical consequences of our program:

```
pw_cc(att,ie,0.976).
pw_cc(att,sn,0.669).
```

Therefore, given a logic program defining lexical and contextual probabilities, we can write another program that uses the automated theorem prover to derive all the required probability facts and constructs a new logic program which is a logical consequence of the original one.

For the contextual module, this is completely straightforward since all possible tags are known in advance. We begin by constructing a program that will prove all the relevant facts by backtracking:

```
con :-
    fc(C1,_),
    fc(C2,_),
    fc(C3,_),
    pc_cc(C1,C2,C3,P),
    write_fact([pc_cc,C1,C2,C3,P]),
    fail.
```

When trying to prove the fact `con`, the theorem prover will first instantiate variables `C1`, `C2` and `C3` to three tags (not necessarily different). It then derives the contextual probability of `C1` given `C2` and `C3`, which becomes the value of the variable `P`, and uses the auxiliary predicate `write_fact/1` to write the fact `pc_cc(`c_1`,`c_2`,`c_3`,`p`)` to the current output stream (where c_1, c_2, c_3 and p are the current values of `C1`, `C2`, `C3` and `P`). The last goal of the body is `fail/0`, which always fails and causes the theorem prover to backtrack until all possible instantiations of `C1`, `C2`, `C3` and `P` have been tried (and all the relevant facts written to the output stream), after which the original goal `con` fails at the top level.

By embedding a call to `con` in the following program, we can construct a database containing all contextual probabilities, stored in an external file:

```
compute_con :-
    telling(Old),
    tell('pc_cc.pl'),
    (con ; true),
    told,
    tell(Old).
```

After setting the output stream to the file `pc_cc.pl`, the program encounters the disjunctive goal `(con ; true)`. As we saw earlier, the first disjunct `con` always fails, but only after all the relevant contextual probability facts have been written to the output stream. The second disjunct `true`, on the other hand, always succeeds, after which the output stream is set back to what it was before the computation started.

In principle, we can use exactly the same method to construct a precompiled lexical module, containing all the relevant facts about lexical probabilities. In practice, however, the situation is complicated by the existence of unknown words. Furthermore, if we want to use the optimized tagger of section 4.2.5, we have to derive several databases corresponding to second-order lexical probabilities, first-order lexical probabilities, and suffix lexical probabilities. Here we will content ourselves with showing how this can be done for the simple baseline tagger from section 4.1 (or for any tagger built only from tools in the basic toolbox of chapter 4). The interested reader is referred to appendix H, which contains the program `tagger.pl`, providing tools for precompiling and loading also the optimized tagger of section 4.2.5.

First of all, we need to split the definition of lexical probabilities into two parts, for known and unknown words:

```
pw_c(W,C,P) :-
     known_pw_c(W,C,P).

pw_c(W,C,P) :-
     \+ known_pw_c(W,_,_),
     unknown_pw_c(C,P).
```

This definition says that the lexical probability of the word `W` given tag `C` is `P` if `P` is the known probability of `W` given `C`, or if `W` lacks known lexical probabilities and `P` is the probability of class `C` being instantiated by an unknown word. This definition presupposes two databases, one for known lexical probabilities, and one for the probabilities of unknown words in open word classes. The following program constructs these databases given a suitable lexical module and frequency databases:

```
compute_lex :-
    telling(Old),
    tell('known_pw_c.pl'),
    ( kpwc ; true ),
    told,
    tell('unknown_pw_c.pl'),
    ( upwc ; true ),
    told,
    tell(Old).

kpwc :-
    fwc(W,C,_),
    pw_c(W,C,P),
    write_fact([known_pw_c,W,C,P]),
    fail.

upwc :-
    open(C),
    unseen_wc(C,F1),
    fc(C,F2),
    P is F1/F2,
    write_fact([unknown_pw_c,C,P]),
```

```
    fail.
```

This program uses exactly the same backtracking technique as the program presented earlier for the contextual module.

4.3.3 Results

Starting from the optimized tagger of section 4.2.5, we modified the search algorithm as described in section 4.3.1 and precompiled the contextual and lexical modules as far as possible using the techniques presented in section 4.3.2. The resulting tagger processed the test corpus in 76.25 seconds, producing exactly the same output as the original tagger, which can be compared with 121.07 seconds for the original tagger (cf. section 4.2.5) and 220.85 seconds for the baseline tagger (cf. section 4.1).

The speed of the final tagger is approximately 150 words per second, which is two orders of magnitude slower than the 14000 words per second reported by [Carlberger and Kann 1999] for a tagger implemented in C++. Nevertheless, 150 words per second is more than adequate for many practical applications, including real-time applications such as dialogue systems.[8] In other words, even though the logic programming tools presented in this report are primarily intended for pedagogical use, they can also be used to build large-scale systems for practical applications.[9]

[8] The rate of human speech is normally less than 10 words per second.

[9] As noted already in the introduction, a system built from essentially this toolbox has been used to tag the 1.2 million word corpus of spoken Swedish collected at the Department of Linguistics, Göteborg University; cf. [Nivre and Grönqvist forthcoming].

Chapter 5

Conclusion

In this thesis, we have presented a toolbox for probabilistic part-of-speech tagging, implemented in a logic programming framework. The tools fall into four groups, corresponding to the four main modules of a probabilistic part-of-speech tagger:

- Tools for the lexical module, i.e., tools for defining and computing probabilities of words conditioned on tags.
- Tools for the contextual module, i.e., tools for defining and computing probabilities of tags conditioned on other tags.
- Search tools, i.e., tools for finding the most probable part-of-speech sequence for a particular word sequence, given the lexical and contextual probability models provided by the lexical and contextual modules.
- Input/Output tools, i.e., tools for pre- and post-processing of textual input and output.

The basic toolbox, which is described in chapter 3, is intended primarily for pedagogical uses and should make it possible for anyone with a rudimentary knowledge of linguistics, probability theory and logic programming to start experimenting with probabilistic part-of-speech tagging. This is to be considered the main contribution of the work presented in this report.

In fact, a subset of the tools presented in this thesis has already been used in teaching, both in a regular course on statistical methods in computational linguistics at Göteborg University, and in a web-based course on statistical natural language processing developed as part of the ELSNET LE Training Showcase (cf. footnote 1, p. 3). In both cases, the experience has been very positive in that students have been able to build working systems very easily and explore different options in a way which clearly deepens the understanding of the concepts involved.

However, we have also tried to show, in chapter 4, how taggers built from the basic toolbox can be enhanced using a variety of different optimization techniques to improve both accuracy and efficiency. In this way, we hope to have shown that the logic programming paradigm provides a viable approach to language technology, not only in the context of traditional knowledge- and grammar-based systems, but also as an implementation tool for probabilistic and

other corpus-based methods. In order to further substantiate this claim, we may note that taggers based on the tools presented in this thesis have already been used successfully in two different research projects. The first, which is reported in [Nivre 2000], is a series of experiments in probabilistic part-of-speech tagging of Swedish texts, with different probabilistic models and different smoothing schemes for both lexical and contextual probabilities. The second, which is described in [Nivre and Grönqvist forthcoming], is the tagging of a large corpus of transcribed spoken Swedish (1.2 million running words). In both cases, it proved very useful to be able to implement and test different versions of a system quickly and easily.

Finally, with a view to the future, we hope that the logical implementation of probabilistic methods may lead to a deeper understanding of the kind of knowledge they incorporate and the kind of inferences they support, a point which is argued further in [Lager and Nivre 1999].

Bibliography

[Baum 1972] Baum, L. E. (1972) An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of a Markov Process. *Inequalities* 3, 1–8.

[Brants and Samuelsson 1995] Brants, T. & Samuelsson, C. (1995) Tagging the Teleman Corpus. In *Proceedings of the 10th Nordic Conference of Computational Linguistics, NODALIDA-95*, Helsinki, 7–20.

[Brill 1992] Brill, E. (1992) A Simple Rule-based Part of Speech Tagger. In *Third Conference of Applied Natural Language Processing*, ACL.

[Brown *et al* 1992] Brown, P. F., Della Pietra, V. J., Jelinek, F., Lafferty, J. D., Mercer, R. L. and Roossin, P. S. (1992) Class-based n-gram models of natural language. *Computational Linguistics* 18, 467–479.

[Carlberger and Kann 1999] Carlberger, J. and Kann, V. (1999) Implementing an Efficient Part-of-Speech Tagger. *Software — Practice and Experience*, 29 (9), 815–832.

[Charniak *et al* 1993] Charniak, E., Hendrickson, C., Jacobson, N. and Perkowitz, M. (1993) Equations for Part-of-Speech Tagging. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI Press/MIT Press.

[Church and Gale 1991] Church, K. W. and Gale, W. A. (1991) A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language* 5, 19–54.

[Covington 1994] Covington, M. A. (1994) *Natural Language Processing for Prolog Programmers.* Englewood Cliffs, NJ: Prentice-Hall.

[Cutting *et al* 1992] Cutting, D., Kupiec, J., Pedersen, J. and Sibun, P. (1992) A Practical Part-of-speech Tagger. In *Third Conference on Applied Natural Language Processing*, ACL, 133–140.

[Daelemans *et al* 1996] Daelemans, W., Zavrel, J., Berck, P. and Gillis, S. (1996) MBT: A Memory-Based Part of Speech Tagger-Generator. In Ejerhed, E. and Dagan, I. (eds) *Proceedings of the Fourth Workshop on Very Large Corpora*, Copenhagen, Denmark, 14-27.

[Dempster *et al* 1977] Dempster, A. P., Laird, N. M. and Rubin, D. B. (1977) Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society* 39, 1–38.

[DeRose 1988] DeRose, S. J. (1988) Grammatical Category Disambiguation by Statistical Optimization. *Computational Linguistics* 14, 31–39.

[Dietterich 1998] Dietterich, T. G. (1998) Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation* 10 (7), 1895–1924.

[Ejerhed *et al* 1992] Ejerhed, E., Källgren, G., Wennstedt, O. and Åström, M. (1992) The Linguistic Annotation System of the Stockholm-Umeå Corpus Project. Report 33. University of Umeå: Department of Linguistics.

[Everitt 1977] Everitt, B. S. (1977) *The Analysis of Contingency Tables*. London: Chapman and Hall.

[Gale and Church 1990] Gale, W. A. and Church, K. W. (1990) Poor Estimates of Context Are Worse Than None. In *Proceedings of the Speech and Natural Language Workshop*, 283–287. Morgan Kaufmann.

[Gale and Church 1994] Gale, W. A. and Church, K. W. (1994) What is wrong with adding one? In Oostdijk, N. and de Haan, P. (eds) *Corpus-Based Research into Language*, 189–198. Amsterdam: Rodopi.

[Gale and Sampson 1995] Gale, W. A. and Sampson, G. (1995) Good-Turing Frequency Estimation Without Tears. *Journal of Quantitative Linguistics* 2, 217–237.

[Good 1953] Good, I. J. (1953) The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrika* 43, 45–63.

[van Halteren *et al* 1998] van Halteren, H., Zavrel, J. and Daelemans, W. (1998) Improving Data Driven Word Class Tagging by System Combination. In *Proceedings of COLING-ACL '98*, Montreal, Canada.

[Jelinek 1990] Jelinek, F. (1990) Self-Organized Language Modeling for Speech Recognition. In Waibel, A. and Lee, K.-F. (eds) *Readings in Speech Recognition*, 450–506. San Mateo, CA: Morgan Kaufmann.

[Jelinek 1997] Jelinek, F. (1997) *Statistical Methods for Speech Recognition.* MIT Press.

[Jelinek and Mercer 1985] Jelinek, F. and Mercer, R. (1985) Probability Distribution Estimation from Sparse Data. *IBM Technical Disclosure Bulletin* 28, 2591–2594.

[Karlsson *et al* 1995] Karlsson, F., Voutilainen, A., Heikkilä, J. and Anttila, A. (eds) *Constraint Grammar. A Language-Independent System for Parsing Unrestricted Text.* Mouton de Gruyter.

[Katz 1987] Katz, S. (1987) Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing* 35, 400–401.

[Koskenniemi 1990] Koskenniemi, K. (1990) Finite-State Parsing and Disambiguation. In *Proceedings of COLING '90*, Helsinki, Finland.

[Lager 1995] Lager, T. (1995) *A Logical Approach to Computational Corpus Linguistics.* Gothenburg Monographs in Linguistics 14. Göteborg University: Department of Linguistics.

[Lager 1999] Lager, T. (1999) The μ-TBL System: Logic Programming Tools for Transformation-Based Learning. In *Proceedings of the Third International Workshop on Computational Natural Language Learning (CoNLL'99)*, Bergen, Norway.

[Lager and Nivre 1999] Lager, T. and Nivre, J. (1999) What Kind of Inference Engines are Part-of-Speech Taggers?. Paper presented at NoDaLiDa 1999, Trondheim, December 9–10, 1999.

[Lidstone 1920] Lidstone, G. J. (1920) Note on the General Case of the Bayes-Laplace Formula for Inductive or *A Posteriori* Probabilities. *Transactions of the Faculty of Actuaries* 8, 182–192.

[Lindberg and Eineborg 1998] Lindberg, N. and Eineborg, M. (1998) Learning Constraint Grammar Style Disambiguation Rules Using Inductive Logic Programming. In *Proceedings of COLING-ACL '98*, Montreal, Canada.

[Lindgren 1993] Lindgren, B. W. (1993) *Statistical Theory.* Chapman-Hall.

[Merialdo 1994] Merialdo, B. (1994) Tagging English Text with a Probabilistic Model. *Computational Linguistics* 20, 155-171.

[Nivre 2000] Nivre, J. (2000) Sparse Data and Smoothing in Statistical Part-of-Speech Tagging. *Journal of Quantitative Linguistics* 7(1), 1–17.

[Nivre and Grönqvist forthcoming] Nivre, J. and Grönqvist, L. (forthcoming) Tagging a Corpus of Spoken Swedish. To appear in *International Journal of Corpus Linguistics.*

[O'Keefe 1990] O'Keefe, R. A. (1990) *The Craft of Prolog.* Cambridge, MA: MIT Press.

[Samuelsson 1994] Samuelsson, C. (1994) Morphological Tagging Based Entirely on Bayesian Inference. In *NODALIDA '93: Proceedings of '9:e Nordiska Datalingistikdagarna', Stockholm, 3–5 June 1993*, 225–238.

[Schmid 1994] Schmid, H. (1994) Part-of-Speech Tagging with Neural Networks. *Proceeding of COLING-94*, 172-176.

[de Smedt *et al* 1999] de Smedt, K., Gardiner, H., Ore, E., Orlandi, T., Short, H., Souillot, J. and Vaughan, W. (1999) *Computing in the Humanities: A European Perspective.* SOCRATES/ERASMUS Thematic Network Project on Advanced Computing in the Humanities. Published by the University of Bergen. [Available at: `http://www.hd.uib.no/AcoHum/book/`.]

[Sterling and Shapiro 1986] Sterling, L. and Shapiro, E. (1986) *The Art of Prolog: Advanced Programming Techniques.* Cambridge, MA: MIT Press.

[Viterbi 1967] Viterbi, A. J. (1967) Error Bounds for Convolutional Codes and an Asymptotically Optimal Decoding Algorithm. *IEEE Transactions on Information Theory* 13, 260–269.

Appendix A

The SUC Tagset

Tag	Part-of-speech	Example (Swedish)
ab	Adverb	*här, sakta*
dl	Delimiter	*., !*
dt	Determiner	*en, varje*
ha	Wh adverb	*var, när*
hd	Wh determiner	*vilken*
hp	Wh pronoun	*vem, vad*
hs	Wh possessive	*vars*
ie	Infinitive marker	*att*
in	Interjection	*hej, usch*
jj	Adjective	*gul, varm*
kn	Conjunction	*och, eller*
nn	Noun	*bil, kvinna*
pc	Participle	*springande*
pl	Particle	*på, an*
pm	Proper name	*Lisa, Växjö*
pn	Pronoun	*hon, dem*
pp	Preposition	*i, på*
ps	Possessive	*min, deras*
rg	Cardinal numeral	*tre, 2000*
ro	Ordinal numberal	*tredje*
sn	Subjunction	*att, eftersom*
uo	Foreign word	*the, yes*
vb	Verb	*går, leka*

Appendix B

Toy Corpus and Frequency Databases

Corpus

```
I/pn can/vb light/vb a/dt fire/nn and/cn you/pn
can/vb open/vb a/dt can/nn of/pp beans/nn ./dl
Now/ab the/dt can/nn is/vb open/jj ,/dl and/cn
we/pn can/vb eat/vb in/pp the/dt light/nn of/pp
the/dt fire/nn ./dl
```

Frequency Databases

```
fwc('.', dl, 2).
fwc(',', dl, 1).
fwc(a, dt, 2).
fwc(and, cn, 2).
fwc(beans, nn, 1).
fwc(can, nn, 2).
fwc(can, vb, 3).
fwc(eat, vb, 1).
fwc(fire, nn, 2).
fwc('I', pn, 1).
fwc(in, pp, 1).
fwc(is, vb, 1).
fwc(light, nn, 1).
fwc(light, vb, 1).
fwc('Now', ab, 1).
fwc(of, pp, 2).
fwc(open, vb, 1).
fwc(open, jj, 1).
fwc(the, dt, 3).
fwc(we, pn, 1).
fwc(you, pn, 1).
```

```
fc(ab, 1).
fc(cn, 2).
fc(dl, 3).
fc(dt, 5).
fc(jj, 1).
fc(nn, 6).
fc(pn, 3).
fc(pp, 3).
fc(vb, 7).

fcc(ab, dt, 1).
fcc(cn, pn, 2).
fcc(dl, ab, 1).
fcc(dl, cn, 1).
fcc(dt, nn, 5).
fcc(jj, dl, 1).
fcc(nn, cn, 1).
fcc(nn, dl, 2).
fcc(nn, pp, 2).
fcc(nn, vb, 1).
fcc(pn, vb, 3).
fcc(pp, dt, 2).
fcc(pp, nn, 1).
fcc(vb, dt, 2).
fcc(vb, jj, 1).
fcc(vb, pp, 1).
fcc(vb, vb, 3).

fccc(ab, dt, nn, 1).
fccc(cn, pn, vb, 2).
fccc(dl, ab, dt, 1).
fccc(dl, cn, pn, 1).
fccc(dt, nn, cn, 1).
fccc(dt, nn, dl, 1).
fccc(dt, nn, pp, 2).
fccc(dt, nn, vb, 1).
fccc(jj, dl, cn, 1).
fccc(nn, cn, pn, 1).
fccc(nn, dl, ab, 1).
fccc(nn, pp, dt, 1).
fccc(nn, pp, nn, 1).
fccc(nn, vb, jj, 1).
fccc(pn, vb, vb, 3).
fccc(pp, nn, dl, 1).
fccc(pp, dt, nn, 2).
fccc(vb, dt, nn, 2).
fccc(vb, jj, dl, 1).
fccc(vb, pp, d1, 1).
fccc(vb, vb, dt, 2).
fccc(vb, vb, pp, 1).
```

Appendix C

Lexical Tools

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.mle.pl
%
% MLE LEXICAL MODEL
%
% Maximum likelihood estimation of lexical probabilities P(w|c).
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the set of open classes (i.e. classes
% allowed for unknown words) is defined by means of clauses of
% the form:
%
% open(C)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_c(W,C,P) :-
        fwc(W,C,F1),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,0) :-
        \+ fwc(W,_,_),
        open(C).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.uni.pl
%
```

```
% UNIFORM SMOOTHING OF LEXICAL MODEL
%
% For known words, the lexical probability is a standard
% maximum likelihood estimate:
%
%    P(w|c) = f(w,c) / f(c)
%
% except that f(c) is adjusted by adding f(c)/n for open
% word classes.
%
% For unknown words, the lexical probability is 1/n
% for all open classes, where n is the number of tokens
% in the training corpus.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
% Finally, it presupposes that the total number of tokens
% in the training corpus is defined:
%
% tokens(N)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_c(W,C,P) :-
        fwc(W,C,F1),
        fc(C,F2),
        \+ open(C),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        fc(C,F2),
        open(C),
        tokens(N),
        P is F1/(F2+(F2/N)).

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
        tokens(N),
```

```
        P is 1/N.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.add.pl
%
% ADDITIVE SMOOTHING OF LEXICAL MODEL
%
% The lexical probability P(w|c) is computed by adding k
% to the frequency f(w,c) and adjusting the class frequencies
% accordingly, i.e.
%
%    P(w|c) = [f(w,c) + k] / [f(c) + kn]
%
% where n is the number of word types.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the additive constant k and the
% number of word types n have been asserted as follows:
%
% lex_add(K)
% types(N)
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
% NB: This lexical model is robust in the sense that every
%     word gets a nonzero probability, even unknown ones.
%     When adjusting the total number of word tokens in a
%     class, the number of word types is taken to be the
%     number of known word types + 1. In other words, all
%     occurrences of unknown words are treated as tokens of
%     a single unknown word type.
%
%     Note also that known words are not allowed to have
%     unknown tags, even though this would normally follow
%     from the addition of k occurrences to the frequency
%     of every word-tag pair.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
pw_c(W,C,P) :-
        fwc(W,C,F1),
        fc(C,F2),
        lex_add(K),
        types(N),
        P is (F1+K)/(F2+(K*(N+1))).

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
        fc(C,F),
        lex_add(K),
        types(N),
        P is K/(F+(K*(N+1))).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.gt.pl
%
% GOOD-TURING SMOOTHING OF LEXICAL MODEL
%
% The lexical probability P(w|c) is defined in terms of
% reestimated frequencies according to Good-Turing's formula.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the reestimated frequencies
% are defined for different categories by clauses of the
% following form:
%
% gtfwc(C,F1,F2)
%
% where C is a word class and F2 is the reestimated frequency
% corresponding to observed frequency for class C.
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_c(W,C,P) :-
        fwc(W,C,F1),
```

```
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
        gtfwc(C,0,F1),
        fc(C,F2),
        P is F1/F2.
```

Appendix D

Contextual Tools

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% con.mle.pl
%
% MLE CONTEXTUAL MODEL
%
% Maximum likelihood estimation of contextual probabilities
% P(c), P(c2|c1), P(c3|c1,c2).
%
% The definitions presuppose that the following
% frequencies are stored in the internal database:
%
% fc(C,F)         - Frequency of class C
% fcc(C1,C2,F)    - Joint frequency of classes C1 and C2
% fccc(C1,C2,C3)  - Joint frequency of classes C1, C2 and C3
%
% NB: fc/2, fcc/3 and fccc/4 must also be defined for the dummy
%     class start used to initialize the tagger. Thus, the
%     following must be defined for all classes C1 and C2:
%
%     fc(start,F).
%     fcc(start,start,F).
%     fcc(start,C1,F).
%     fccc(start,start,C1,F).
%     fccc(start,C1,C2,F).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% P(c)

pc(C,P) :-
        fc(C,F),
        tokens(N),
        P is F/N.
```

```
% P(c2|c1)

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        fc(C1,F2),
        P is F1/F2.

pc_c(C2,C1,0) :-
        \+ fcc(C1,C2,_).

% P(c3|c1,c2)

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        fcc(C1,C2,F2),
        P is F1/F2.

pc_cc(C3,C1,C2,0) :-
        \+ fccc(C1,C2,C3,_).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% con.add.pl
%
% ADDITIVE SMOOTHING OF CONTEXTUAL MODEL
%
% The contextual probabilities P(c2|c1) and P(c3|c1,C2) are
% computed by adding k to the frequencies f(c1,c2) and
% f(c1,c2,c3) and adjusting the total frequencies accordingly,
% i.e.
%
%    P(c2|c1) = [f(c1,c2) + k] / [f(c1) + kn]
%    P(c3|c1,c2) = [f(c1,c2,c3) + k] / [f(c1,c2) + k(n**2)]
%
% where n is the number of classes.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fc(C,F)           - Frequency of class C
% fc(C1,C2,F)       - Joint frequency of class C1 and C2
% fc(C1,C2,C3,F)    - Joint frequency of class C1, C2 and C3
%
% NB: fc/2, fcc/3 and fccc/4 must also be defined for the dummy
%     class start used to initialize the tagger. Thus, the
%     following must be defined for all classes C1 and C2:
%
%     fc(start,F).
%     fcc(start,start,F).
```

```
%     fcc(start,C1,F).
%     fccc(start,start,C1,F).
%     fccc(start,C1,C2,F).
%
% It also presupposes that the additive constant k, the number
% of tokens m and the number of classes n have been asserted
% as follows:
%
% lex_add(K)
% tokens(M)
% classes(N)
%
% NB: This model is robust in the sense that every class
%     gets a non-zero probability in every context. However,
%     variables will not bind to unseen classes, so in order
%     for an unseen class C to get a probability, the first
%     argument has to be instantiated to C when calling
%     pc_c/3 or pc_cc/4. (This instantiation will normally
%     be achieved by a call to pw_c/3.)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% P(c)

pc(C,P) :-
        fc(C,F),
        tokens(N),
        P is F/N.

% P(c2|c1)

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        fc(C1,F2),
        con_add(K),
        classes(N),
        P is (F1+K)/(F2+(K*N)).

pc_c(C2,C1,P) :-
        \+ fcc(C1,C2,_),
        fc(C1,F),
        con_add(K),
        classes(N),
        P is K/(F+(K*N)).

% P(c3|c1,c2)

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        fcc(C1,C2,F2),
```

```
        con_add(K),
        classes(N),
        P is (F1+K)/(F2+(K*N*N)).

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        fcc(C1,C2,F),
        con_add(K),
        classes(N),
        P is K/(F+(K*N*N)).

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        \+ fcc(C1,C2,_),
        con_add(K),
        classes(N),
        P is K/(K+(K*N)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% con.gt.pl
%
% GOOD-TURING ESTIMATION OF CONTEXTUAL MODEL
%
% The contextual probabilities P(c2|c1) and P(c3|c1,C2) are
% computed by using the reestimated biclass and triclass
% frequencies according to Good-Turing's formula.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fc(C,F)           - Frequency of class C
% fc(C1,C2,F)       - Joint frequency of class C1 and C2
% fc(C1,C2,C3,F)    - Joint frequency of class C1, C2 and C3
%
% NB: fc/2, fcc/3 and fccc/4 must also be defined for the dummy
%     class start used to initialize the tagger. Thus, the
%     following must be defined for all classes C1 and C2:
%
%     fc(start,F).
%     fcc(start,start,F).
%     fcc(start,C1,F).
%     fccc(start,start,C1,F).
%     fccc(start,C1,C2,F).
%
% It also presupposes that the reestimated frequencies
% are defined in a database with clauses of the following form:
%
% gtfcc(F1,F2) (biclass)
```

```
% gtfccc(F1,F2) (triclass).
%
% where F2 is the reestimated frequency corresponding to the
% observed frequency F1.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% P(c)

pc(C,P) :-
        fc(C,F),
        tokens(N),
        P is F/N.

% P(c2|c1)

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        gtfcc(F1,F2),
        fc(C1,F3),
        P is F2/F3.

pc_c(C2,C1,P) :-
        \+ fcc(C1,C2,_),
        gtfcc(0,F1),
        fc(C1,F2),
        P is F1/F2.

% P(c3|c1,c2)

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        gtfccc(F1,F2),
        fcc(C1,C2,F3),
        gtfcc(F3,F4),
        P is F2/F4.

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        gtfccc(0,F1),
        fcc(C1,C2,F2),
        gtfcc(F2,F3),
        P is F1/F3.

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        gtfccc(0,F1),
        \+ fcc(C1,C2,_),
        gtfcc(0,F2),
        P is F1/F2.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% con.back.pl
%
% BACKOFF SMOOTHING OF CONTEXTUAL MODEL
%
% The contextual probabilities P(c2|c1) and P(c3|c1,c2) are
% estimated as follows:
%
%    P(c2|c1) =    (1-d)*(f(c1,c2)/f(c1))   if f(c1,c2) > t
%                  a*P(c2)         otherwise
%
%    P(c3|c1,c2) = (1-d)*(f(c1,c2,c3)/f(c1,c2)
%                                  if f(c1,c2,c3) > t
%                  a*P(c3|c2)      otherwise
%
% where a = 1-d if the f(context) > 0 and a = 1 otherwise.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fc(C,F)           - Frequency of class C
% fc(C1,C2,F)       - Joint frequency of class C1 and C2
% fc(C1,C2,C3,F)    - Joint frequency of class C1, C2 and C3
%
% NB: fc/2, fcc/3 and fccc/4 must also be defined for the dummy
%     class start used to initialize the tagger. Thus, the
%     following must be defined for all classes C1 and C2:
%
%     fc(start,F).
%     fcc(start,start,F).
%     fcc(start,C1,F).
%     fccc(start,start,C1,F).
%     fccc(start,C1,C2,F).
%
% It also presupposes that the threshold t, the discounting
% factor d, the number of tokens m and the number of
% classes n have been asserted as follows:
%
% threshold(T)
% discount(D)
% tokens(M)
% classes(N)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% P(c)
```

```
pc(C,P) :-
        fc(C,F),
        tokens(N),
        P is F/N.

% P(c2|c1)

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        threshold(T),
        F1 > T,
        fc(C1,F2),
        discount(D),
        P is (1-D)*F1/F2.

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        threshold(T),
        \+ F1 > T,
        pc(C2,P1),
        discount(D),
        P is D*P1.

pc_c(C2,C1,P) :-
        \+ fcc(C1,C2,_),
        pc(C2,P1),
        discount(D),
        P is D*P1.

% P(c3|c1,c2)

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        threshold(T),
        F1 > T,
        fcc(C1,C2,F2),
        discount(D),
        P is (1-D)*F1/F2.

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        threshold(T),
        \+ F1 > T,
        pc_c(C3,C2,P1),
        discount(D),
        P is D*P1.

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        fcc(C1,C2,_),
```

```
        pc_c(C3,C2,P1),
        discount(D),
        P is D*P1.

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        \+ fcc(C1,C2,_),
        pc_c(C3,C2,P).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% con.int.pl
%
% SMOOTHING OF CONTEXTUAL MODEL BY LINEAR INTERPOLATION
%
% The contextual probabilities P(c2|c1) and P(c3|c1,c2) are
% estimated by a the following weighted sum of ML estimates:
%
%    P(c2|c1) = k_1 P(c2) + k_2 P(c2|c1)   [k1+k2=1]
%    P(c3|c1,c2) = k_1 P(c2) + k_2 P(c2|c1) + k_3 P(c3|c1,c2)
%                                           [k1+k2+k3=1]
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fc(C,F)           - Frequency of class C
% fc(C1,C2,F)       - Joint frequency of class C1 and C2
% fc(C1,C2,C3,F)    - Joint frequency of class C1, C2 and C3
%
% NB: fc/2, fcc/3 and fccc/4 must also be defined for the dummy
%     class start used to initialize the tagger. Thus, the
%     following must be defined for all classes C1 and C2:
%
%     fc(start,F).
%     fcc(start,start,F).
%     fcc(start,C1,F).
%     fccc(start,start,C1,F).
%     fccc(start,C1,C2,F).
%
% It also presupposes that the constants k_i, the number
% of tokens m and the number of classes n have been asserted
% as follows:
%
% bi1(K1)
% bi2(K2)
% tri1(K1)
% tri2(K2)
% tri3(K3)
% tokens(M)
```

```
% classes(N)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% P(c)

pc(C,P) :-
        fc(C,F),
        tokens(N),
        P is F/N.

% P(c2|c1)

pc_c(C2,C1,P) :-
        fcc(C1,C2,F1),
        fc(C1,F2),
        pc(C2,P1),
        bi1(K1),
        bi2(K2),
        P is K1*P1+K2*F1/F2.

pc_c(C2,C1,P) :-
        \+ fcc(C1,C2,_),
        pc(C2,P1),
        bi1(K1),
        P is K1*P1.

% P(c3|c1,c2)

pc_cc(C3,C1,C2,P) :-
        fccc(C1,C2,C3,F1),
        fcc(C1,C2,F2),
        pc(C3,P1),
        pc_c(C3,C2,P2),
        tri1(K1),
        tri2(K2),
        tri3(K3),
        P is K1*P1+K2*P2+K3*F1/F2.

pc_cc(C3,C1,C2,P) :-
        \+ fccc(C1,C2,C3,_),
        pc(C3,P1),
        pc_c(C3,C2,P2),
        tri1(K1),
        tri2(K2),
        P is K1*P1+K2*P2.
```

Appendix E

Search Tools

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% viterbi.bi.pl
%
% Viterbi biclass tagging, by Joakim Nivre, based on a
% program by Torbjörn Lager. Last modified 18 July 2000.
%
% Main predicate: most_probable_path(Words,P-Path)
% where Words is a list of words (to be tagged),
% Path is the optimal path, i.e. word class sequence,
% and P is the probability of Path.
%
% The tagger relies on two types of probabilities
% being defined elsewhere:
%
% pw_c(Word,Class,Prob) - the probability of Word given Class,
%                         i.e. P(Word|Class)
%
% pc_c(Class2,Class1,Prob) - the probability of observing
%                            Class2 after Class1,
%                            i.e. P(Class2|Class1)
%
% NB: Initial probabilities are given as: pc_c(Class,start,P).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%:- use_module(library(lists)).  NB: Needed in SISCtus Prolog

most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start]],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start|Path]).


paths([],MPPs,MPPs).
```

```
paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_c(Word,Class2,LexP),
            findall(P1-[Class2,Class1|RestPath],
                    (member(P0-[Class1|RestPath],PPs0),
                    pc_c(Class2,Class1,ConP),
                    P1 is LexP*ConP*P0),
                PPs1),
           most_probable(PPs1,PP)),
        PPs2),
    paths(Words,PPs2,PPs).


most_probable([P-Path|PPs],MPP) :-
       most_probable(PPs,P-Path,MPP).

most_probable([],MPP,MPP).
most_probable([P-Path|PPs],HP-BPath,MPP) :-
       (   P > HP ->
           most_probable(PPs,P-Path,MPP)
       ;   most_probable(PPs,HP-BPath,MPP)
       ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% viterbi.tri1.pl
%
% Viterbi triclass tagging, by Joakim Nivre, based on a
% program by Torbjörn Lager. Last modified 18 July 2000.
% Bug fixed 18 November 1999 (thanks to James Cussens).
%
% Main predicate: most_probable_path(Words,P-Path)
% where Words is a list of words (to be tagged),
% Path is the optimal path, i.e. word class sequence,
% and P is the probability of Path.
%
% The tagger relies on two types of probabilities
% being defined elsewhere:
%
% pw_c(Word,Class,Prob) - the probability of Word given Class,
%                         i.e. P(Word|Class)
%
% pc_cc(Class3,Class1,Class2,Prob) - the probability of
%                          observing Class3 after Class1 and
%                          Class2, i.e. P(Class3|Class1,Class2)
%
% NB: Initial probabilities are given as:
%     pc_cc(Class,start,start,P).
%     Post-initial probabilities as:
```

```
%     pc_cc(Class2,start,Class1,P).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start,start]],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start,start|Path]).

paths([],MPPs,MPPs).
paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_c(Word,Class3,LexP),
            pc(Class2,_), %get Class2s
            findall(P1-[Class3,Class2,Class1|RestPath],
                    (member(P0-[Class2,Class1|RestPath],PPs0),
                    pc_cc(Class3,Class1,Class2,ConP),
                    P1 is LexP*ConP*P0),
                PPs1),
           most_probable(PPs1,PP)),
        PPs2),
    paths(Words,PPs2,PPs).

most_probable([P-Path|PPs],MPP) :-
       most_probable(PPs,P-Path,MPP).

most_probable([],MPP,MPP).
most_probable([P-Path|PPs],HP-BPath,MPP) :-
       (   P > HP ->
           most_probable(PPs,P-Path,MPP)
       ;   most_probable(PPs,HP-BPath,MPP)
       ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% viterbi.tri2.pl
%
% Implementation of Viterbi tagging for second-order
% Markov models, by Joakim Nivre.
% Last modified 18 July 2000.
%
% Main predicate: most_probable_path(Words,P-Path)
% where Words is a list of words (to be tagged),
% Path is the optimal path, i.e. state sequence,
% and P is the probability of Path.
%
% The tagger relies on two types of probabilities
% being defined elsewhere:
%
```

```
% pw_cc(Word,Class1,Class2,Prob) - the probability of
%       Word given Class1 (preceding tag) and Class2
%       (tag of Word), i.e. P(Word|Class1,Class2)
%
% pc_cc(Class3,Class1,Class2,Prob) - the probability of
%       Class3 given Class1 and Class2,
%       i.e. P(Class3|Class1,Class2)
%
% NB: Initial probabilities are given as:
%     pc_cc(C,start,start,P).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start,start]],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start,start|Path]).

paths([],MPPs,MPPs).
paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_cc(Word,Class2,Class3,LexP),
             findall(P1-[Class3,Class2,Class1|RestPath],
                     (member(P0-[Class2,Class1|RestPath],PPs0),
                      pc_cc(Class3,Class1,Class2,ConP),
                      P1 is P0*LexP*ConP),
                     PPs1),
             most_probable(PPs1,PP)),
            PPs2),
    paths(Words,PPs2,PPs).

most_probable([P-Path|PPs],MPP) :-
    most_probable(PPs,P-Path,MPP).

most_probable([],MPP,MPP).
most_probable([P-Path|PPs],HP-BPath,MPP) :-
       (   P > HP ->
           most_probable(PPs,P-Path,MPP)
       ;   most_probable(PPs,HP-BPath,MPP)
       ).
```

Appendix F

Input/Output Tools

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% io.pl
%
% Input/output module for probabilistic part-of-speech tagging
% by Joakim Nivre.
% Last modified 7 July 2000.
%
% The basic predicates are
% tag/0 - interactive tagging, both input and output from/to user
% tag/1 - tag input file, output to user
% tag/2 - tag input file (arg1), output to file (arg2)
%
% For file input, the tagger presupposes a format with one word
% per line (no annotation) and looks for major delimiters (.,?,!)
% to segment the input. (Without such segmentation, underflow
% errors may occur if the string to tag becomes too long.)
%
% File output is printed in the format
% Word<tab>Tag
% with one word per line.
%
% NB: The tagger presupposes that the predicate
%     most_probable_path/2 is defined (see e.g. viterbi.gen.pl).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Basic tagging procedure(s)

tag :-
        % default input is 'user'.
        tag(user).

tag(InputFile) :-
        % default output is 'user'.
```

```
        tag(InputFile,user).


tag(InputFile,OutputFile) :-
        seeing(OldInputFile),
        telling(OldOutputFile),
        see(InputFile),
        tell(OutputFile),
        repeat,
                read_string(Words,EndOfFile),
                ( most_probable_path(Words,_P-Path) ->
                  write_tagged_string(Words,Path) ;
                  write_string(Words) ),
                EndOfFile == yes,
        !,
        seen,
        told,
        see(OldInputFile),
        tell(OldOutputFile).


% Reading a string ending in '.', '!' or '?' from input stream

read_string(Words,Eof) :-
        get0(C),
        read_string(C,Words,Eof).

read_string(-1,[],/*End of File*/yes) :- !.

read_string(10,Words,Eof) :-
        !,
        read_string(Words,Eof).

read_string(32,Words,Eof) :-
        !,
        read_string(Words,Eof).

read_string(C,[Word|Words],Eof) :-
        read_rest_letters(Chars,LeftOver,Eof),
        atom_chars(Word,[C|Chars]),
        (\+ delimiter(Word) -> read_string(LeftOver,Words,Eof)
        ; Words=[]).

read_rest_letters(Letters,LeftOver,Eof) :-
        get0(C),
        read_rest_letters(C,Letters,LeftOver,Eof).

read_rest_letters(-1,[],-1,yes) :- !.
read_rest_letters(10,[],10,_) :- !.
read_rest_letters(32,[],32,_) :- !.
```

```
read_rest_letters(C,[C|Chars],LeftOver,Eof) :-
        read_rest_letters(Chars,LeftOver,Eof).

delimiter('.').
delimiter('?').
delimiter('!').


% Writing strings

write_tagged_string([],[]).
write_tagged_string([Word|Words],[Tag|Tags]) :-
        write(Word),put(9),write(Tag),nl,
        write_tagged_string(Words,Tags).

write_string([Word|Words]) :-
        write(Word),nl,
        write_string(Words).
```

Appendix G

Configuration Tools

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% main.pl
%
% Program for configuring probabilistic taggers.
% The following subdirectories and files are presupposed:
%
% DIRECTORY   FILES       DESCRIPTION
%
% lex/        lex.mle.pl  MLE lexical model
%             lex.uni.pl  MLE with uniform for unknown words
%             lex.add.pl  MLE with additive smoothing
%             lex.gt.pl   Good-Turing estimation
%             lex.uni.pl  MLE with uniform for unknown words
%             fwc.pl      Lexical frequency db [fwc(W,C,F).]
%             open.pl     Database of open classes [open(C).]
%             gtfwc.pl    Reestimated freqs for GTE (lexical)
%
% con/        con.mle.pl  MLE lexical model
%             con.add.pl  MLE with additive smoothing
%             con.gt.pl   Good-Turing estimation
%             con.back.pl Backoff smoothing
%             con.int.pl  Linear interpolation
%             fc.pl       Tag frequency db [fc(C,F).]
%             fcc.pl      Tag bigram freq db [fcc(C1,C2,F).]
%             fccc.pl     Tag trigram freq db [fccc(C1,C2,C3,F).]
%             gtfcc.pl    Reestimated freqs for GTE (biclass)
%             gtfccc.pl   Reestimated freqs for GTE (triclass)
%
% viterbi/    viterbi.bi.pl   Viterbi biclass
%             viterbi.tri1.pl Viterbi triclass (standard)
%             viterbi.tri2.pl Viterbi second-order model
%
% io/         io.pl           Input/output module
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%:- use_module(library(lists)).    % Uncomment these two lines if
%:- use_module(library(charsio)).  % you are using SICStus Prolog

user_message :-
    nl, nl,
    write('WARNING: Before you can use the tagger,'), nl,
    write('you must choose system parameters by'), nl,
    write('executing the command ''setup''.'),
    nl, nl, nl.

:- user_message.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Utility predicates
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sum_list([],0).         % Remove these four lines if you are
sum_list([X|Xs],N) :-   % using SICStus Prolog
    sum_list(Xs,M),     %
    N is X+M.           %

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Setup Procedure
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

setup :-
    nl,
    write('Tagging model (bi, tri1, tri2): '),
    read(Model),
    ( Model = bi -> load_biclass ;
      ( Model = tri1 -> load_triclass1 ;
          (Model = tri2 -> load_triclass2 ))),
    abolish(pc,2),
    abolish(pc_c,3),
    abolish(pc_cc,4),
    abolish(con_add,1),
    abolish(bi1,1),
    abolish(bi2,1),
    abolish(tri1,1),
    abolish(tri2,1),
    abolish(tri3,1),
    abolish(discount,1),
    abolish(threshold,1),
    abolish(gtfcc,2),
```

```
    abolish(gtfccc,2),
    nl,
    write('Smoothing for contextual model'), nl,
    write('(none, add, int, back, gt): '),
    read(CSmoothing),
    ( CSmoothing = none -> load_con_mle ;
      ( CSmoothing = add -> load_con_add ;
        ( CSmoothing = int -> load_con_int(Model) ;
          ( CSmoothing = back -> load_con_back ;
            ( CSmoothing = gt -> load_con_gt(Model) ))))),
    abolish(pw_c,3),
    abolish(lex_add,1),
    abolish(gtfwc,3),
    nl,
    write('Smoothing for lexical model'), nl,
    write('(none, add, uni, gt): '),
    read(LSmoothing),
    ( LSmoothing = none -> load_lex_mle ;
      ( LSmoothing = add -> load_lex_add ;
        ( LSmoothing = uni -> load_lex_uni ;
          (LSmoothing = gt -> load_lex_gt )))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Install Markov model
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


load_biclass :-
        abolish(most_probable_path,2),
        abolish(paths,3),
        abolish(most_probable,2),
        abolish(most_probable,3),
        consult('viterbi/viterbi.bi.pl').

load_triclass1 :-
        abolish(most_probable_path,2),
        abolish(paths,3),
        abolish(most_probable,2),
        abolish(most_probable,3),
        consult('viterbi/viterbi.tri1.pl').

load_triclass2 :-
        abolish(most_probable_path,2),
        abolish(paths,3),
        abolish(most_probable,2),
        abolish(most_probable,3),
        consult('viterbi/viterbi.tri2.pl').
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Install contextual model
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- consult('con/fc.pl').
:- consult('con/fcc.pl').
:- consult('con/fccc.pl').

load_con_add :-
    write('Additive constant: '),
    read(K),
    assert(con_add(K)),
    consult('con/con.add.pl').

load_con_mle :- consult('con/con.mle.pl').

load_con_int(bi) :-
    write('Weight for uniclass model: '),
    read(K1),
    K2 is 1-K1,
    K3 is K2-K1,
    write('Weight for biclass model = '),
    write(K2),
    nl,
    assert(bi1(K1)),
    assert(bi2(K2)),
    assert(tri1(K1)),
    assert(tri2(K1)),
    assert(tri3(K3)),
    consult('con/con.int.pl').

load_con_int(Model) :-
    \+ Model = bi,
    write('Weight for uniclass model: '),
    read(K1),
    write('Weight for biclass model: '),
    read(K2),
    K3 is 1-(K1+K2),
    K4 is 1-K1,
    write('Weight for triclass model = '),
    write(K3),
    nl,
    assert(tri1(K1)),
    assert(tri2(K2)),
    assert(tri3(K1)),
    assert(bi1(K1)),
    assert(bi2(K4)),
    consult('con/con.int.pl').
```

```
load_con_back :-
    write('Frequency threshold: '),
    read(T),
    assert(threshold(T)),
    write('Discounting factor: '),
    read(D),
    assert(discount(D)),
    consult('con/con.back.pl').

load_con_gt(Model) :-
    consult('con/con.gt.pl'),
    consult('con/gtfcc.pl'),
    ( \+ Model = bi -> consult('con/gtfccc.pl') ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Install lexical model
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- consult('lex/fwc.pl').
:- abolish(tokens,1).
:- abolish(types,1).
:- abolish(classes,1).

classes :-
    setof(C,F^fc(C,F),Cs),
    length(Cs,N),
    assert(classes(N)).

tokens :-
    bagof(F,W^C^fwc(W,C,F),Fs),
    sum_list(Fs,N),
    assert(tokens(N)).

types :-
    setof(W,C^F^fwc(W,C,F),Ws),
    length(Ws,N),
    assert(types(N)).

start_state :-
    assert((fc(start,F) :- tokens(F))),
    assert((fcc(start,C,F) :- fc(C,F))),
    assert((fccc(start,C1,C2,F) :- fcc(C1,C2,F))).

% NB: Remove start_state and store rules directly in
% fc.pl, fcc.pl and fccc.pl if using SICStus Prolog

:- tokens.
```

```
:- types.
:- classes.
:- start_state.
:- consult('lex/open.pl').

load_lex_mle :-
    consult('lex/lex.mle.pl').

load_lex_add :-
    write('Additive constant: '),
    read(K),
    assert(lex_add(K)),
    consult('lex/lex.add.pl').

load_lex_uni :-
    consult('lex/lex.uni.pl').

load_lex_gt :-
    consult('lex/lex.gt.pl'),
    consult('lex/gtfwc.pl').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Install I/O
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- consult('io/io.pl').
```

Appendix H

Optimization Tools

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.gt.num.pl
%
% GOOD-TURING SMOOTHING OF LEXICAL MODEL
%
% The lexical probability P(w|c) is defined in terms of
% reestimated frequencies according to Good-Turing's formula.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the reestimated frequencies
% are defined for different categories by clauses of the
% following form:
%
% gtfwc(C,F1,F2)
%
% where C is a word class and F2 is the reestimated frequency
% corresponding to observed frequency for class C.
%
% In addition, it presupposes that the reestimated frequency of
% zero-frequency words is defined for different categories
% by clauses of the following form:
%
% unseen(C,F)
%
% where F is the reestimated frequency of zero-frequency words
% of class C.
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
```

```
% clauses of the following form:
%
% open(C)
%
% Unknown words that can be parsed as numerals using the
% predicate numeral/1, defined in num.pl are only allowed
% to have the tag 'rg' (numeral).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- consult('num.pl'). % Change path to 'num.pl' if using
                      % SICStus Prolog

pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,rg,P) :-
        \+ fwc(W,_,_),
        numeral(W),
        unseen_wc(rg,F1),
        fc(rg,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        \+ numeral(W),
        open(C),
        unseen_wc(C,F1),
        fc(C,F2),
        P is F1/F2.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% num.pl
%
% PARSING OF SWEDISH NUMERALS
%
% numeral(Word) is true if Word can be completely decomposed
% into numeral morphemes ("en", "tusen", etc.) or if Word begins
```

```
% with a digit (0-9) and do not contain any letters.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

num("en").
num("ett").
num("två").
num("tre").
num("fyra").
num("fem").
num("sex").
num("sju").
num("åtta").
num("nio").
num("nie").
num("tio").
num("tie").
num("elva").
num("tolv").
num("tretton").
num("fjorton").
num("femton").
num("sexton").
num("sjutton").
num("arton").
num("nitton").

num("biljoner").
num("biljon").
num("miljarder").
num("miljard").
num("miljoner").
num("miljon").

num("tusen").
num("hundra").
num("hunra").

num("tjugo").
num("tjuge").
num("tju").
num("trettio").
num("tretti").
num("fyrtio").
num("förtio").
num("förti").
num("femti").
num("femtio").
num("sextio").
num("sexti").
```

```
num("sjuttio").
num("sjutti").
num("åttio").
num("åtti").
num("nittio").
num("nitti").

num("noll").

numeral(W) :-
        atom_chars(W,[C|Cs]),
        47 < C, C < 58,
        !,
        no_letters(Cs).

numeral(W) :-
    atom_chars(W,Cs),
    nums(Cs).

nums([]).
nums(S) :-
    split(S,S1,S2),
    num(S2),
    nums(S1).

split(S,S,[]).
split(S,[],S).
split([A|S],[A|T],U) :- split(S,T,U).

no_letters([]).
no_letters([C|Cs]) :-
        \+ letter(C),
        no_letters(Cs).

letter(C) :-
        C > 64, C < 91.
letter(C) :-
        C > 96, C < 123.
letter(196).
letter(197).
letter(214).
letter(228).
letter(229).
letter(246).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.gt.cap.pl
%
```

```
% GOOD-TURING SMOOTHING OF LEXICAL MODEL
%
% The lexical probability P(w|c) is defined in terms of
% reestimated frequencies according to Good-Turing's formula.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the reestimated frequencies
% are defined for different categories by clauses of the
% following form:
%
% gtfwc(C,F1,F2)
%
% where C is a word class and F2 is the reestimated frequency
% corresponding to observed frequency for class C.
%
% In addition, it presupposes that the reestimated frequency of
% zero-frequency words is defined for different categories
% by clauses of the following form:
%
% unseen(C,F)
%
% where F is the reestimated frequency of zero-frequency words
% of class C.
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
% For unknown words, the lexical probabilities are adjusted
% if the word is capitalized and/or the tag involved is
% proper name.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
```

```
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
        unseen_wc(C,F1),
        fc(C,F2),
        P1 is F1/F2,
        ( cap(W) -> ( \+C==pm -> P is 0.02*P1 ; P is P1 ) ;
          ( C==pm -> P is 0.05*P1 ; P is P1) ).

cap(W) :-
        atom_chars(W,[C1,C2|_Cs]),
        uppercase(C1),
        \+ uppercase(C2).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% io.cap.pl
%
% Input/output module for probabilistic part-of-speech tagging
% by Joakim Nivre.
% Last modified 7 July 2000.
%
% The basic predicates are
% tag/0 - interactive tagging, both input and output from/to user
% tag/1 - tag input file, output to user
% tag/2 - tag input file (arg1), output to file (arg2)
%
% For file input, the tagger presupposes a format with one word
% per line (no annotation) and looks for major delimiters (.,?,!)
% to segment the input. (Without such segmentation, underflow
% errors may occur if the string to tag becomes too long.)
% If the first word of a string to be tagged is capitalized
% and unknown, it is decapitalized before the string is passed
% to the tagger.
%
% File output is printed in the format
% Word<tab>Tag
% with one word per line.
%
% NB: The tagger presupposes that the predicate
%     most_probable_path/2 is defined (see e.g. viterbi.gen.pl).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Basic tagging procedure(s)

tag :-
        % default input is 'user'.
        tag(user).

tag(InputFile) :-
        % default output is 'user'.
        tag(InputFile,user).


tag(InputFile,OutputFile) :-
        seeing(OldInputFile),
        telling(OldOutputFile),
        see(InputFile),
        tell(OutputFile),
        abolish(runtime,1),
        assert(runtime(0)),
        repeat,
                read_string(Words,EndOfFile),
                decap(Words,DWords),
                ( most_probable_path(DWords,_P-Path) ->
                  write_tagged_string(Words,Path) ;
                  write_string(Words) ),
                EndOfFile == yes,
        !,
        seen,
        told,
        see(OldInputFile),
        tell(OldOutputFile).


% Reading a string ending in '.', '!' or '?' from input stream

read_string(Words,Eof) :-
        get0(C),
        read_string(C,Words,Eof).

read_string(-1,[],/*End of File*/yes) :- !.

read_string(10,Words,Eof) :-
        !,
        read_string(Words,Eof).

read_string(32,Words,Eof) :-
        !,
        read_string(Words,Eof).

read_string(C,[Word|Words],Eof) :-
        read_rest_letters(Chars,LeftOver,Eof),
```

```
        atom_chars(Word,[C|Chars]),
        (\+ delimiter(Word) -> read_string(LeftOver,Words,Eof)
        ; Words=[]).

read_rest_letters(Letters,LeftOver,Eof) :-
        get0(C),
        read_rest_letters(C,Letters,LeftOver,Eof).

read_rest_letters(-1,[],-1,yes) :- !.
read_rest_letters(10,[],10,_) :- !.
read_rest_letters(32,[],32,_) :- !.
read_rest_letters(C,[C|Chars],LeftOver,Eof) :-
        read_rest_letters(Chars,LeftOver,Eof).

delimiter('.').
delimiter('?').
delimiter('!').


% Writing strings

write_tagged_string([],[]).
write_tagged_string([Word|Words],[Tag|Tags]) :-
        write(Word),put(9),write(Tag),nl,
        write_tagged_string(Words,Tags).

write_string([Word|Words]) :-
        write(Word),nl,
        write_string(Words).


% Decapitalizing a word

decap([Word|Words],[DWord|Words]) :-
        \+ fwc(Word,_,_),
        !,
        atom_chars(Word,Cs),
        decapitalize(Cs,Ds),
        atom_chars(DWord,Ds).

decap(W,W).

decapitalize([],[]).
decapitalize([C|Cs],[D|Ds]) :-
        ( uppercase(C) -> D is C+32 ;
          D is C ),
        decapitalize(Cs,Ds).

uppercase(C) :- C > 64, C < 91.
uppercase(196).
```

```
uppercase(197).
uppercase(214).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.gt.suf.pl
%
% GOOD-TURING SMOOTHING OF LEXICAL TRICLASS MODEL
%
% Lexical probabilities P(w|c) is defined by the predicate pw_c/3
% in terms of reestimated frequencies according to Good-Turing's
% formula.
%
% The definitions presuppose that the following
% frequencies are stored in the internal database:
%
% fwcc(W,C1,C2,F) - Joint frequency of word W and class C2
%                   preceded by class C1
% fwc(W,C,F)      - Joint frequency of word W and class C
% fc(C,F)         - Frequency of class C
%
% It also presupposes that the reestimated frequencies
% are defined for different categories by clauses of the
% following form:
%
% gtfwc(C,F1,F2)
%
% where C is a word class and F2 is the reestimated frequency
% corresponding to observed frequency for class C.
%
% In addition, it presupposes that the reestimated frequency of
% zero-frequency words is defined for different categories
% by clauses of the following form:
%
% unseen(C,F)
%
% where F is the reestimated frequency of zero-frequency words
% of class C.
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
% For unknown words, we try to find the longest
% suffix occurring in the training corpus and use the lexical
% probability defined by the tag-suffix frequencies. If no
% suffix is found, the reestimated zero frequencies for open
```

```
% classes are used.
%
% This treatment of unknown words presupposes that tag-suffix
% frequencies are stored in the internal database:
%
% fsc(S,C,F) - Joint frequency of suffix S and class C
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        unknown_prob(W,C,P).

unknown_prob(W,C,P) :-
        ending(E,W),
        (fwc(E,C,F1) ; fsc(E,C,F1)),
        open(C),
        fc(C,F2),
        P is F1/F2.

unknown_prob(W,C,P) :-
        \+ ending(_,W),
        open(C),
        unseen_wc(C,F1),
        fc(C,F2),
        P is F1/F2.

ending(E,W) :-
        atom_chars(W,Ws),
        suffix(Es,Ws),
        atom_chars(E,Es),
        fsc(E,_,_),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
% lex.gt.tri.pl
%
% GOOD-TURING SMOOTHING OF LEXICAL MODEL
%
% The lexical probability P(w|c) is defined in terms of
% reestimated frequencies according to Good-Turing's formula.
%
% The definition presupposes that the following
% frequencies are stored in the internal database:
%
% fwc(W,C,F)    - Joint frequency of word W and class C
% fc(C,F)       - Frequency of class C
%
% It also presupposes that the reestimated frequencies
% are defined for different categories by clauses of the
% following form:
%
% gtfwc(C,F1,F2)
%
% where C is a word class and F2 is the reestimated frequency
% corresponding to observed frequency for class C.
%
% In addition, it presupposes that the reestimated frequency of
% zero-frequency words is defined for different categories
% by clauses of the following form:
%
% unseen(C,F)
%
% where F is the reestimated frequency of zero-frequency words
% of class C.
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


pw_cc(W,C1,C2,P) :-
        fwcc(W,C1,C2,F1),
        hfreq(F),
        F1 > F,
        fcc(C1,C2,F2),
        P is F1/F2.

pw_cc(W,C1,C2,P) :-
        pw_c(W,C2,P),
        fc(C1,_),
```

```
        \+ (fwcc(W,C1,C2,F1), hfreq(F), F1 > F).

pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        open(C),
        unseen_wc(C,F1),
        fc(C,F2),
        P is F1/F2.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lex.gt.opt.pl
%
% GOOD-TURING SMOOTHING OF LEXICAL TRICLASS MODEL
%
% If the frequency f(w_i,c_(i-1),c_i) is above a pre-defined
% threshold, the probability P(w_i|c_(i-1),c_i) is defined by
% the predicate pw_cc/4, using pure MLE.
%
% If the frequency is below the threshold, the lexical
% probability P(w|c) is defined by the predicate pw_c/3
% in terms of reestimated frequencies according to Good-Turing's
% formula.
%
% The definitions presuppose that the following
% frequencies are stored in the internal database:
%
% fwcc(W,C1,C2,F) - Joint frequency of word W and class C2
%                   preceded by class C1
% fwc(W,C,F)      - Joint frequency of word W and class C
% fc(C,F)         - Frequency of class C
%
% It also presupposes that the reestimated frequencies
% are defined for different categories by clauses of the
% following form:
%
```

```
% gtfwc(C,F1,F2)
%
% where C is a word class and F2 is the reestimated frequency
% corresponding to observed frequency for class C.
%
% In addition, it presupposes that the reestimated frequency of
% zero-frequency words is defined for different categories
% by clauses of the following form:
%
% unseen(C,F)
%
% where F is the reestimated frequency of zero-frequency words
% of class C.
%
% Finally, it presupposes that the number of open classes
% (i.e. those allowed for unknown words) are defined by
% clauses of the following form:
%
% open(C)
%
% Unknown words that can be parsed as numerals using the
% predicate numeral/1, defined in num.pl are only allowed
% to have the tag 'rg' (numeral).
%
% For other unknown words, we first try to find the longest
% suffix occurring in the training corpus and use the lexical
% probability defined by the tag-suffix frequencies. If no
% suffix is found, the reestimated zero frequencies for open
% classes are used. In both cases, the lexical probabilities
% are adjusted if the word is capitalized and/or the tag
% involved is proper name.
%
% This treatment of unknown words presupposes that tag-suffix
% frequencies are stored in the internal database:
%
% fsc(S,C,F) - Joint frequency of suffix S and class C
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- consult('num.pl'). % Change path to 'lex.num.pl' if using
                       % SICStus Prolog

pw_cc(W,C1,C2,P) :-
        fwcc(W,C1,C2,F1),
        F1 > 10,
        fcc(C1,C2,F2),
        P is F1/F2.

pw_cc(W,C1,C2,P) :-
        pw_c(W,C2,P),
```

```
        fc(C1,_),
        \+ (fwcc(W,C1,C2,F1), hfreq(F), F1 > F).

pw_c(W,C,P) :-
        fwc(W,C,F1),
        \+ open(C),
        fc(C,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        fwc(W,C,F1),
        open(C),
        gtfwc(C,F1,F2),
        fc(C,F3),
        P is F2/F3.

pw_c(W,rg,P) :-
        \+ fwc(W,_,_),
        numeral(W),
        unseen_wc(rg,F1),
        fc(rg,F2),
        P is F1/F2.

pw_c(W,C,P) :-
        \+ fwc(W,_,_),
        \+ numeral(W),
        unknown_prob(W,C,P1),
        ( cap(W) -> ( \+C==pm -> P is 0.02*P1 ; P is P1 ) ;
          ( C==pm -> P is 0.05*P1 ; P is P1) ).

unknown_prob(W,pm,P) :-
        cap(W),
        unseen_wc(pm,F1),
        fc(pm,F2),
        P is F1/F2.

unknown_prob(W,C,P) :-
        ending(E,W),
        (fwc(E,C,F1) ; fsc(E,C,F1)),
        open(C),
        fc(C,F2),
        P is F1/F2.

unknown_prob(W,C,P) :-
        \+ ending(_,W),
        open(C),
        unseen_wc(C,F1),
        fc(C,F2),
        P is F1/F2.
```

```
cap(W) :-
        atom_chars(W,[C1,C2|_Cs]),
        uppercase(C1),
        \+ uppercase(C2).

ending(E,W) :-
        atom_chars(W,Ws),
        suffix(Es,Ws),
        atom_chars(E,Es),
        fsc(E,_,_),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% viterbi.tri3.pl
%
% Implementation of Viterbi tagging for second-order
% Markov models, by Joakim Nivre.
% Last modified 18 July 2000.
%
% Main predicate: most_probable_path(Words,P-Path)
% where Words is a list of words (to be tagged),
% Path is the optimal path, i.e. state sequence,
% and P is the probability of Path.
%
% The tagger relies on two types of probabilities
% being defined elsewhere:
%
% pw_cc(Word,Class1,Class2,Prob) - the probability of Word
%       given Class1 (preceding tag) and Class2 (tag of Word),
%       i.e. P(Word|Class1,Class2)
%
% pc_cc(Class3,Class1,Class2,Prob) - the probability of
%       Class3 given Class1 and Class2,
%       i.e. P(Class3|Class1,Class2)
%
% NB: Initial probabilities are given as: pc_cc(C,start,start,P).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start,start]],[start],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start,start|Path]).

paths([],MPPs,_,MPPs).
paths([Word|Words],PPs0,Classes0,PPs) :-
    findall(PP,
            (member(Class2,Classes0),
```

```
            pw_cc(Word,Class2,Class3,LexP),
            findall(P1-[Class3,Class2,Class1|RestPath],
                    (member(P0-[Class2,Class1|RestPath],PPs0),
                     pc_cc(Class3,Class1,Class2,ConP),
                     P1 is P0*LexP*ConP),
                    PPs1),
            most_probable(PPs1,PP)),
           PPs2),
    setof(Class,P^Rest^member(P-[Class|Rest],PPs2),Classes2),
    paths(Words,PPs2,Classes2,PPs).

most_probable([P-Path|PPs],MPP) :-
    most_probable(PPs,P-Path,MPP).

most_probable([],MPP,MPP).
most_probable([P-Path|PPs],HP-BPath,MPP) :-
       (   P > HP ->
           most_probable(PPs,P-Path,MPP)
       ;   most_probable(PPs,HP-BPath,MPP)
       ).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% tagger.pl
%
% Tools for precompiling and and loading taggers built from
% the basic toolbox and optimization tools.
%
% In order to precompile a tagger, call the predicate
% build_tagger/1 as follows:
%
%    build_tagger(base).   - Baseline tagger (section 4.1)
%    build_tagger(opt).    - Optimized tagger (section 4.2.5)
%
% In order to load a precompiled tagger, call the predicate
% load_tagger/1 with argument base or opt.
%
% The following directories and files are presupposed:
%
% DIRECTORY   FILES       DESCRIPTION
%
% lex/        lex.gt.pl   Good-Turing estimation
%             fwcc.pl     Lexical frequency db [fwcc(W,C1,C2,F).]
%             fwc.pl      Lexical frequency db [fwc(W,C,F).]
%             fsc.pl      Suffix frequenct db [fsc(S,C,F).]
%             open.pl     Database of open classes [open(C).]
%             gtfwc.pl    Reestimated freqs for GTE (lexical)
%             unknown_pw_c.pl  Definition of unknown lexical
%                              probabilities for optimized tagger
```

```
%
% con/        con.add.pl  MLE with additive smoothing
%             fc.pl       Tag frequency db [fc(C,F).]
%             fcc.pl      Tag bigram freq db [fcc(C1,C2,F).]
%             fccc.pl     Tag trigram freq db [fccc(C1,C2,C3,F).]
%             gtfcc.pl    Reestimated freqs for GTE (biclass)
%             gtfccc.pl   Reestimated freqs for GTE (triclass)
%
% viterbi/    viterbi.tri4.pl Viterbi triclass (standard)
%             viterbi.tri3.pl Viterbi second-order model
%
% io/         io.pl           Input/output module (standard)
%             io.opt.pl       Input/output module (optimized)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%:- use_module(library(lists)).   Uncomment these lines if using
%:- use_module(library(charsio)). SICStus Prolog

load_tagger(Version) :-
        load_viterbi(Version),
        load_io(Version),
        load_con_db(Version),
        load_lex_db(Version).

build_tagger(Version) :-
        load_con(Version),
        load_lex(Version),
        compute_con(Version),
        compute_lex(Version).

load_viterbi(Version) :-
        abolish(most_probable_path,2),
        abolish(paths,3),
        abolish(most_probable,2),
        abolish(most_probable,3),
        load_viterbi_file(Version).

load_viterbi_file(opt) :-
        consult('viterbi/viterbi.tri3.pl').

load_viterbi_file(base) :-
        consult('viterbi/viterbi.tri4.pl').

load_io(Version) :-
        abolish(tag,0),
        abolish(tag,1),
        abolish(tag,2),
        abolish(read_string,2),
        abolish(read_string,3),
```

```
        abolish(read_rest_letters,3),
        abolish(read_rest_letters,4),
        abolish(delimiter,1),
        abolish(write_tagged_string,2),
        abolish(write_string,1),
        abolish(decap,2),
        abolish(decapitalize,2),
        abolish(uppercase,1),
        load_io_file(Version).

load_io_file(opt) :-
        consult('io/io.opt.pl').

load_io_file(base) :-
        consult('io/io.pl').

load_con_db(_) :-
        abolish(tags,1),
        abolish(fc,2),
        abolish(fcc,3),
        abolish(fccc,4),
        abolish(pc,2),
        abolish(pc_c,3),
        abolish(pc_cc,4),
        consult('con/tags.pl'),
        consult('con/pc_cc.pl').

load_lex_db(Version) :-
        abolish(fwc,3),
        abolish(fwcc,4),
        abolish(fsc,3),
        abolish(gtfwc,3),
        abolish(pw_cc,4),
        abolish(pw_c,3),
        abolish(known_pw_cc,4),
        abolish(known_pw_c,3),
        abolish(unknown_pw_c,3),
        abolish(ps_c,3),
        abolish(unseen_pw_c,2),
        abolish(cap,1),
        abolish(ending,2),
        load_lex_db_file(Version).

load_lex_db_file(opt) :-
        consult('lex/num.pl'),
        consult('lex/open.pl'),
        consult('lex/unknown_pw_c.pl'),
        consult('lex/unseen_pw_c.pl'),
        consult('lex/pw_cc.pl'),
        consult('lex/pw_c.pl'),
```

```
        consult('lex/ps_c.pl').

load_lex_db_file(base) :-
        consult('lex/open.pl'),
        consult('lex/unseen_pw_c.pl'),
        consult('lex/pw_c.pl').

load_con(_) :-
    abolish(tags,1),
    abolish(con_add,1),
    abolish(fc,2),
    abolish(fcc,3),
    abolish(fccc,4),
    abolish(pc,2),
    abolish(pc_c,3),
    abolish(pc_cc,4),
    assert(con_add(0.5)),
    consult('con/con.add.pl'),
    consult('con/fc.pl'),
    consult('con/fcc.pl'),
    consult('con/fccc.pl').

load_lex(Version) :-
    abolish(known_pw_cc,4),
    abolish(known_pw_c,3),
    abolish(unknown_pw_c,3),
    abolish(ps_c,3),
    abolish(unseen_pw_c,2),
    abolish(cap,1),
    abolish(ending,2),
    abolish(open,1),
    abolish(tokens,1),
    abolish(types,1),
    abolish(classes,1),
    abolish(fwc,3),
    abolish(fwcc,4),
    abolish(fsc,3),
    abolish(gtfwc,3),
    abolish(unseen_wc,2),
    abolish(pw_c,3),
    abolish(pw_cc,4),
    load_lex_file(Version),
    tokens,
    types,
    classes.

load_lex_file(opt) :-
    consult('lex/lex.gt.pl'),
    consult('lex/num.pl'),
    consult('lex/open.pl'),
```

```
    consult('lex/gtfwc.pl'),
    consult('lex/fwc.pl'),
    consult('lex/fwcc.pl'),
    consult('lex/fsc.pl').

load_lex_file(base) :-
    consult('lex/lex.gt.pl'),
    consult('lex/open.pl'),
    consult('lex/gtfwc.pl'),
    consult('lex/fwc.pl').

tokens :-
    bagof(F,W^C^fwc(W,C,F),Fs),
    sum_list(Fs,N),
    assert(tokens(N)).

types :-
    setof(W,C^F^fwc(W,C,F),Ws),
    length(Ws,N),
    assert(types(N)).

classes :-
    setof(C,F^fc(C,F),Cs),
    length(Cs,N),
    assert(classes(N)).

compute_con(_) :-
    telling(Old),
    tell('con/pc_cc.pl'),
    (con ; true),
    told,
    tell('con/tags.pl'),
    (tags ; true),
    told,
    tell(Old).

con :-
    fc(C1,_),
    fc(C2,_),
    fc(C3,_),
    pc_cc(C1,C2,C3,P),
    write_fact([pc_cc,C1,C2,C3,P]),
    fail.

tags :-
    fc(C1,_),
    write_fact([is_tag,C1]),
    fail.

compute_lex(opt) :-
```

```
    telling(Old),
    tell('lex/pw_cc.pl'),
    abolish(hfreq,1),
    assert(hfreq(10)),
    write('pw_cc(W,C1,C2,P) :- known_pw_cc(W,C1,C2,P).'), nl,
    write('pw_cc(W,C1,C2,P) :- known_pw_c(W,C2,P),
           \\+ known_pw_cc(W,C1,C2,_).'), nl,
    write('pw_cc(W,_,C2,P) :- \\+ known_pw_c(W,_,_),
           unknown_pw_c(W,C2,P).'), nl,
    ( kpwcc ; true ),
    told,
    tell('lex/pw_c.pl'),
    ( kpwc ; true ),
    told,
    tell('lex/ps_c.pl'),
    ( psc ; true ),
    told,
    tell('lex/unseen_pw_c.pl'),
    ( upwc ; true ),
    told,
    tell(Old).

compute_lex(base) :-
    telling(Old),
    tell('lex/pw_c.pl'),
    write('pw_c(W,C,P) :- known_pw_c(W,C,P).'), nl,
    write('pw_c(W,C,P) :- \\+ known_pw_c(W,_,_),
          unseen_pw_c(C,P).'), nl,
    ( kpwc ; true ),
    told,
    tell('lex/unseen_pw_c.pl'),
    ( upwc ; true ),
    told,
    tell(Old).

kpwcc :-
    fwcc(W,C1,C2,F1),
    hfreq(F2),
    F1 > F2,
    fcc(C1,C2,F3),
    P is F1/F3,
    write_fact([known_pw_cc,W,C1,C2,P]),
    fail.

kpwc :-
    fwc(W,C,_),
    pw_c(W,C,P),
    write_fact([known_pw_c,W,C,P]),
    fail.
```

```
psc :-
    fsc(E,C,F1),
    fc(C,F2),
    P is F1/F2,
    write_fact([ps_c,E,C,P]),
    fail.

upwc :-
    open(C),
    unseen_wc(C,F1),
    fc(C,F2),
    P is F1/F2,
    write_fact([unseen_pw_c,C,P]),
    fail.

write_fact([F|Args]) :-
        write(F),
        write('('),
        write_args(Args).

write_args([X]) :-
        !,
        writeq(X),
        write(').'),
        nl.

write_args([X|Xs]) :-
        writeq(X),
        write(','),
        write_args(Xs).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unknown_pw_c.pl
%
% Definition of lexical probabilities for unknown words
% for optimized tagger (section 4.2.5).
%
% The following predicates are presupposed:
%
%    numeral/1        cf. num.pl, section 4.2.1
%    unseen_pw_c/2    generated by build_tagger(opt)
%    known_pw_c/3     generated by build_tagger(opt)
%    ps_c/3           generated by build_tagger(opt)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
unknown_pw_c(W,rg,P) :-
        numeral(W),
        !,
        unseen_pw_c(rg,P).

unknown_pw_c(W,C,P) :-
        unknown_prob(W,C,P1),
        ( cap(W) -> ( \+C==pm -> P is 0.02*P1 ; P is P1 ) ;
          ( C==pm -> P is 0.05*P1 ; P is P1 ) ).

unknown_prob(W,pm,P) :-
        cap(W),
        unseen_pw_c(pm,P).

unknown_prob(W,C,P) :-
        suffix_prob(W,C,P).

suffix_prob(W,C,P) :-
        ending(E,W),
        !,
        (known_pw_c(E,C,P) ; ps_c(E,C,P)),
        open(C).

suffix_prob(_,C,P) :-
        open(C),
        unseen_pw_c(C,P).

cap(W) :-
        atom_chars(W,[C1,C2|_Cs]),
        uppercase(C1),
        \+ uppercase(C2).

ending(E,W) :-
        atom_chars(W,Ws),
        suffix(Es,Ws),
        atom_chars(E,Es),
        ps_c(E,_,_),
        !.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% viterbi.tri4.pl
%
% Viterbi triclass tagging, by Joakim Nivre, based on a
% program by TorbjĄörn Lager. Last modified 18 July 2000.
% Bug fixed 18 November 1999 (thanks to James Cussens).
% Special version for use with precompiled lexical and
% contextual modules generated by build_tagger(base).
%
```

```
% Main predicate: most_probable_path(Words,P-Path)
% where Words is a list of words (to be tagged),
% Path is the optimal path, i.e. word class sequence,
% and P is the probability of Path.
%
% The tagger relies on two types of probabilities
% being defined elsewhere:
%
% pw_c(Word,Class,Prob) - the probability of Word given Class,
%                         i.e. P(Word|Class)
%
% pc_cc(Class3,Class1,Class2,Prob) - the probability of
%                   observing Class3 after Class1 and Class2,
%                   i.e. P(Class3|Class1,Class2)
%
% NB: Initial probabilities are given as:
%                   pc_cc(Class,start,start,P).
%     Post-initial probabilities as:
%                   pc_cc(Class2,start,Class1,P).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

most_probable_path(Words,P-Path) :-
    paths(Words,[1-[start,start]],PPs),
    most_probable(PPs,P-Path1),
    reverse(Path1,[start,start|Path]).

paths([],MPPs,MPPs).
paths([Word|Words],PPs0,PPs) :-
    findall(PP,
            (pw_c(Word,Class3,LexP),
            is_tag(Class2), %get Class2s
            findall(P1-[Class3,Class2,Class1|RestPath],
                    (member(P0-[Class2,Class1|RestPath],PPs0),
                    pc_cc(Class3,Class1,Class2,ConP),
                    P1 is LexP*ConP*P0),
                PPs1),
           most_probable(PPs1,PP)),
         PPs2),
    paths(Words,PPs2,PPs).

most_probable([P-Path|PPs],MPP) :-
       most_probable(PPs,P-Path,MPP).

most_probable([],MPP,MPP).
most_probable([P-Path|PPs],HP-BPath,MPP) :-
       (   P > HP ->
           most_probable(PPs,P-Path,MPP)
       ;   most_probable(PPs,HP-BPath,MPP)
       ).
```